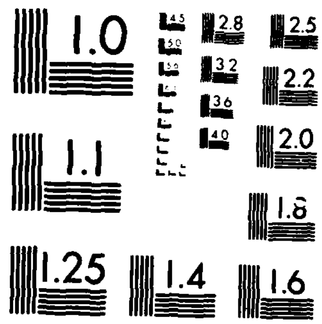
AD-A164 255 STRUCTURED MICROCONTROLLER DESIGN USING PLA FIRMWARE 1/1
(U) CINCINNATI UNIV OH C A PAPACHRISTOU 12 DEC 85
ARO-18626.12-EL DAAG29-82-K-0106

UNCLASSIFIED F/G 9/2 NL

END
FILMED
DTIC




MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

ARO 18626.12-EL

(2)

AD-A164 255

STRUCTURED MICROCONTROLLER DESIGN USING

PLA FIRMWARE


Christos A. Papachristou


FINAL REPORT

December 12, 1985

U.S. ARMY RESEARCH OFFICE


Contract No: DAAG29-82-K-0106



University of Cincinnati

Case Western Reserve University




APPROVED FOR PUBLIC RELEASE;

DISTRIBUTION UNLIMITED.

86 2 ? 191

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

# REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

| | | |
|---|---|---|
| 1. REPORT NUMBER<br>ARO 18626.12-EL | 2. GOVT ACCESSION NO.<br>N/A | 3. RECIPIENT'S CATALOG NUMBER<br>N/A |
| 4. TITLE (and Subtitle)<br>Structured Microcontroller Design Using PLA Firmware | | 5. TYPE OF REPORT & PERIOD COVERED<br>FINAL 04/19/82 to 10/18/85 |
| | | 6. PERFORMING ORG. REPORT NUMBER |
| 7. AUTHOR(s)<br>Christos A. PAPACHRISTOU | | 8. CONTRACT OR GRANT NUMBER(s)<br>DAAG29-82-K-0106 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS<br>University of Cincinnati, Cincinnati, Ohio 45221<br>Case Western Reserve University, Cleveland Ohio, 44106 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS<br>N/A |
| 11. CONTROLLING OFFICE NAME AND ADDRESS<br>U. S. Army Research Office<br>Post Office Box 12211<br>Research Triangle Park, NC 27709 | | 12. REPORT DATE<br>December 12, 1985 |
| | | 13. NUMBER OF PAGES<br>42 |
| 14. MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office) | | 15. SECURITY CLASS. (of this report)<br>Unclassified |
| | | 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

NA

18. SUPPLEMENTARY NOTES

The view, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Microprogramming, Firmware Engineering, Microcontrol Architecture, Micro-Simulation, PLA Structures, VLSI, Microcode Development, CAD Tools

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Please see reverse side

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT CONTINUED


## ABSTRACT

The objective of this research is to contribute a design methodology for microprogramming architectures with supporting firmware and development tools. Two PLA-based microcontrol architectures have been proposed that are suitable for modular microprogramming. The first scheme consists of a PLA sequence store, a microcode ROM and an address processor. This structure has the capability of complex microsequencing such as multiway branching, microsubroutines, nested microlooping and the like. To alleviate the pin-limitation problem, a bit-slice approach is taken in the second scheme which allows for easy microcontrol expandability and compaction of the sequence store.

Firmware support for the microcontrollers is provided by such control constructs as if-then-else, while-do and the like, which are available at the microlevel. Several firmware design tools have been developed and incorporated into a software package, MMDS, a Modular Microprogram Development System. MMDS includes the following tools: a microcode assembler, a microsequencer assembler, a PLA code formatter and a functional-level simulator of modular microarchitectures.

An automatic migration approach based on these tools has been successfully initiated. Several compaction techniques for VLSI microcode have been implemented. A hardware design language for microarchitecture definition has also been developed and tested. Integration of these tools will provide a design environment for implementation of VLSI microcode.

TABLE OF CONTENTS

LIST OF FIGURES, TABLES AND APPENDIXES

# I. INTRODUCTION

Microprogramming is an elegant technique to systematically structure the control section of computers and digital systems. Although it was introduced [1] during the first generation computers, it was not adopted commercially until the third generation computers [2], due to memory limitations. Today, microprogramming is widespread from large mainframes to small microsystems, although it has evolved substantially since its early conception. Microprogram memories are available just like main memories, as writable control stores. This distinguishes user-oriented microprogrammable systems from microprogrammed machines [3]. Microprogramming is an important method for interpretation and emulation [4] of computer systems. Microprogramming is also a promising technique for vertical migration of operating system primitives and other complicated software to firmware [5-6].

Modern integrated circuit technology has affected microprogramming by means of hardware control devices such as ROMs, PLAs and microsequencers [7]. ROMs are used as control memories, PLAs for efficient address mapping and microsequencers are useful elements to implement control functions. Although these devices have been available as discrete LSI chips, they are now basic components of VLSI [8]. Most 16 and 32 bit microprocessors contain such devices occupying large chip areas. With increasing demand for complicated control sequencing in VLSI, there is a growing need for modularity and structure in both microprogram architectures and firmware code, and there is also need for development aids. Some work in this area has already appeared in the literature. Schemes for microprogram multiway branching have been investigated in [9-10]. Structures for modular microprogram sequencing have been proposed in [11-12].

The research reported in this report has three objectives. First, to propose a hardware architecture of a complex microcontrol scheme suitable for modular microprogramming. Second, to develop firmware support by means of primitive and compound constructs that allow complex control sequencing. Third, to construct firmware design and development tools such as microsequencer and microcode assemblers and simulators for the benefit of the user. The motivation for this work is the need to migrate in firmware not just traditional microprograms but also more complex software functions, for example parsers or even operating systems, to improve speed, reliability, security and the like.

This report is organized as follows. Section II describes the proposed microcontrol scheme. The structured firmware support for the microcontroller is provided in Section III. An expandable, "bit-slice" modification of the microcontroller is given in Section IV. Several firmwares design tools are briefly described in Sections V and VI. A test example of firmware implementation, a binary search tree algorithm, is demonstrated in Section VII. Significant results on automatic migration, VLSI implementation and a hardware design language tool for microcode design are summarized in Section VIII. Concluding remarks are in Section IX.

# II. MICROSEQUENCER ARCHITECTURE

## 2.1 Structure of the Microcontrol System

This section describes the architecture of a complex microsequencer suitable for modular microprogramming. The basic objective is to implement compound sequencing functions that facilitate block-structured firmware development. This includes efficient address modification to enable modular multiway branching, modular looping, microsubroutine nesting, microcoroutines and the like.

The microcontroller scheme shown in Fig.1 consists of three basic components: the PLA sequencing store, the address processor and the microcode store; the latter may consist of PLAs or ROMs. Recall that there are two fundamental tasks with every microprogram control scheme: microsequencing and microcoding [13]. This information, normally embedded within each microinstruction, is separated in our approach which is reflected in the PLA and the ROM stores, respectively. This technique provides more capability for the compound sequencing mentioned earlier.

It is important to note that the sequencing PLA is utilized as a read-only associative memory [14] to store the microsequencing information required. This is an advantage with respect to the overall storage requirements, to be detailed shortly. The microcode store contains the required control information embedded as formatted micro-opcodes. The address processor generates the addressing information for the sequencing and microcode stores. Some details of the sequencer and address processor follow.

## 2.2 Sequence Store

This unit stores all the sequencing information for the microcode store in the form of firmware constructs or transactions. Each transaction occupies one PLA word and consists of three fields: (a) the input address field, (b) the microsequencing function field, and (c) a branch code or addressing field. The input addressing field contains the address of the current transaction to be matched by the effective address, for both the sequence store, and control store carried on the address bus (Fig.I). The function field contains encoded sequencing information by means of directives or commands for the address processor. The main sequencing functions to be implemented by this scheme are listed in Table 1 with comments. The third field contains either branching information for the address processor (multiple jump addresses) or branch code indicating the (multiple) status conditions to be tested in case of multiway jumps.

In microprogrammed memories, sequentially executable microinstructions are stored in sequential addresses. Thus, there is a need of an implicit addressing scheme of the form:

NEXT ADDRESS = PRESENT ADDRESS + 1

together with ways of explicit address generation in case of "JUMPS". In our approach, transaction for implicit sequencing are labeled CONTINUE-type

whereas the other transactions are labeled JUMP-type. It is important to note that there is no need to store CONTINUE-type of transactions in the sequence store. This information is implicitly conveyed to the address processor, using the PLA as an associative memory element. Only JUMP-type of transactions need to be stored in the sequence store.

### 2.3 Address Processor

The address processor, Fig. 2, may be viewed as a primitive, special purpose CPU with the PLA serving as its memory. The processor operates on an input "data" stream, i.e., addresses fetched from the sequencer PLA, or an external source, under the control of an "instruction", i.e. the function code from the sequencer PLA. The basic functional control elements and data storage elements of the address processor, shown in Fig. 2, are (a) address modifier, (b) PLA controller, (c) address stack, (d) branch code stack and (e) address multiplexer. Other hardware modules required include status-testers, encoders, data multiplexer and a adder.

The PLA controller generates the control signals for the address processor from the "FUNCTION" field of the sequencer PLA. The address multiplexer is used to select the addressing information from either the address modifier or an externally mapped address. The address stack is used for linkage and contains return addresses enabling nested microsubroutine calls and nested loops. Each word in this stack consists of two fields: (a) RTN-CODE and (b) RTN-ADDRESS. The return code essentially parameterizes the return function to enable a variety of return functions. The branch code stack stores encoded branch information regarding status conditions to be tested in case of multiway branch type of sequencing transactions.

## III. BLOCK STRUCTURED FIRMWARE

### 3.1 Firmware Blocks

In this section we discuss the organization and the structuring of the sequencing transactions of Table I that provide firmware support for the hardware control scheme of Section II. To facilitate modular microprogramming, firmware implementation of these constructs is based on the concept of firmware block or module, i.e. a sequence of microinstructions with single entry and exit points. Further, a firmware block is context free, i.e. independent of its location, and additionally, it can contain conditional or unconditional calls to other blocks. Thus, a block structured microprogram consists of a listing of such formatted constructs as the ones in Table I which control the sequencing of firmware blocks.

The following notation is introduced to facilitate transaction formatting. Let X, Y, Z, ..... denote labels of transactions and let F, G, H, ..... denote labels of firmware blocks. Let bF, sF and eF denote the labels of the first, second and last transaction, respectively, of block F. Decimal numerals may also be appended to extent this notation, e.g. bF1, eF1, etc. By prefixing A and N to the previous labels we designate the current and next addresses, respectively, of the transaction in reference.

### 3.2 Organization of Microprogram Memories

The microsequencer architecture proposed directly supports a micromemory addressing space of up to 64K words. With an arbitrary organization of firware blocks in the control memory, up to sixteen bit addressing information would be required to address a block. In multiway modular transactions with many address fields, this may be a serious limitation. To alleviate this problem, the sequence and microcode stores are organized to enable zero-page addressing in all modular transactions. In this scheme, the address of the first transaction of each firware block must be in page zero i.e. in the first 256 locations in the micromemory address space.

The organization of microprogram memories with two firware blocks F and G is shown in Fig.3. The first microinstruction of module F is stored at absolute address one in the microcode ROM. Correspondingly, at absolute input address one in the sequence store, a JUMP sequencing function is stored containing sF, the sixteen bit absolute address of the second microinstruction of module F in the microcode store. Thus, concurrent to the execution of the first microinstruction of module F, the absolute address of the second microinstruction is loaded into the microprogram counter of Fig. 2 by the JUMP sequencing function.

This addressing scheme for firmware blocks results in a decreased length of the addressing subfields in the sequencer. Thus, only eight bits are required to address an arbitrary module.

### 3.3 Transaction Formulation and Formatting

Every transaction is associated with a sequencing action by means of a Function-Code such as CALL, DLOOP, MAP, etc. (see Table I). Specifically, a transaction consists of three fields, namely, the (current) address, the function code and the branch code or address fields, consistent with the format of the PLA sequence store of Section II, denoted as follows:

/Address//Function Code/Branch Code or Next Address(es)/

The 'm' modifying addresses, in case of 'm'-way direct module branching, would be represented by bF1, bF2, ..., bFm. The Branch-Code field in case of a SBC transaction would contain a sequence of decimal values C1, C2, ... ,Cn indicating the external status signals to be tested. In case of DLOOP transactions, the third field would contain a decimal value indicating the number of times the iteration is to be performed, while in case of RTN and MAP transactions, the third field would be absent. The interpretation of the third field depends on the function codes. As illustrations we have:

```
/AX//MJUMP/MX1,MX2,MX3,MX4/ ;Multiway intra-module branching
/AX//MCALL/bF1,bF2,bF3/     ;Multiway modular branching
/AY//SBC/C1,C2,C3,C4,C5/    ;Store branch codes
/AZ//CALL/NX=bF/            ;Unconditional call
```

The sequencing transactions of Table I are basic and compound types. The basic transactions are conceptually similar to the fundamental constructs of structured programming ,i.e., they are (a) sequential (if-then),

(b) conditional (if-then-else), (c) iterative (loop) and (d) case-like to allow multiway branching. The formatting of these constructs in firware is shown in Fig. 4, (a)-(d). More details are given in [15]. On the basis of these constructions, compound sequencing constructs can also be developed. An example is the modular loop of Table I, MLOOP, whose formatting is shown in Fig. 5. This is a quite useful transaction, to be demonstrated later.

We remark that the SBC transaction should precede all multiway sequencing transactions prefixed by M, e.g. MCALL, as shown in Figs. 3 and 4. The equate (=) symbol above is used for explicit address or condition assignments. Also, the blocks F,G, etc. in the same figures have the same structure. The different returns are parameterized by the return code stored along with the return addresses in the address stack of Fig. 2, thus maintaining the context free property of firmware.

The implementation of the above firware structures at the microlevel is aided by several user-oriented tools, discussed later.

## IV. BIT-SLICE MICROCONTROLLER

### 4.1 Rationale

A single chip implementation of the address processor constrains the branching capability of the microcontrol scheme. Multiway branching requires a corresponding number of address fields located in the same PLA word. Thus, due to pin-count constraints, there is a limitation on the number of branch addresses that can be accomodated for direct multiway branching. Time multiplexing the addresses on the single bus, to reduce the pin-count, involves time overhead. Another solution would be to implement both the processor and the sequence store within a VLSI chip to reduce external communication. This scheme was considered in [16] but it seemed suitable for more customized designs. We have taken instead a bit-slice approach to modify the design of the address processor (AP). This approach has the following advantages: (a) solves the pin limitation problem, (b) allows for easy expandability, and (c) results in compaction of the sequence store.

In contrast to the conventional bit-sliced microcontrol designs [17], the slices in the address processor are not uniform. Thus, a fully expanded address processor consists of one primary slice (module) and one or more (identical) secondary slices. It should be noted, though, that this "slicing" of the processor requires the partitioning of the sequence store into corresponding PLA "slices" to accomodate the AP slices. We shall discuss the overall system organization after first describing the AP sliced structure.

### 4.2 Primary and Secondary Slices

The architecture of the primary slice is shown in Fig. 6a. The major control and data processing elements are: (a) PLA controller, (b) adder, (c) address stack, (d) branch code stack, (g) status tester and (h) multiplexers.

The PLA controller (inside the AP) generates the control signals for the address processor, depending on the input received from the function-opcode field of the PLA sequencer and the internal AP status. The microprogram counter is used as an addressing element for both the control and sequence stores. There is a 2's complement 16-bit adder whose right and left inputs are selected by MUX 1 and 2, respectively. The control signals for MUX 1 and 2 are generated by the priority condition selector and the (internal) PLA controller. The input selections of MUX 1 and 2 are shown in Fig. 6a with more details being given in [16].

The status tester is used for testing the external status signals in case of conditional sequencing transactions. If the status signals are not mutually exclusive, the priority condition selector will resolve the conflict. The address stack is used for microsubroutine linking and microlooping. It is 18-bit x 8 words, allowing for a nesting of up to 8-levels, and it includes a 16-bit return address and a 2-bit return code which parameterizes the return. The Branch-Code stack is 16-bits x 8 words and is used to store the branch code conditions to be tested in case of multiway modular calls and looping. The loop stack is 8-bits x 8 words and is used in count-down type iterations for up to 8 levels of nesting and with a maximum count of 255. The transaction stack is 16-bits x 8 words deep. It is used in modular looping transactions.

The architecture of the secondary slice is shown in Fig. 6b. The internal control signals are also generated by the PLA controller. The primary slice itself gives the sequencer a capability of direct three-way branching which is further increased by two for each additional secondary slice used in the expansion. The secondary slice either outputs the eight LSBs (least significant bits) from the inputs received in the address field, or the eight MSBs, or the output bus is tristated, depending on the priority condition selector and the chip enable (CE) signals. The other elements of the secondary slice serve the same purpose as in the primary.

### 4.3 System Organization and Compaction

A typical control unit using the above address processor slices, PLAs as sequence store and ROM for control memory, is shown in Fig. 7. The 16-bit primary slice output serves as the addressing input to the control ROM and the PLAs of the corresponding processor slices. The PLAs required include, first, the function opcode PLA, i.e. a 16-bit input, 4-bit output PLA used for storing the sequencing function field of each transaction (see Table I). Recall that sequencing information regarding CONTINUE type microinstructions is not stored in the sequence store since this is implicitly generated by default in the function-opcode PLA. In addition, with each address processor slice a 16-bit input, 16-bit output PLA is required. These PLAs are utilized to store the branch-code or address subfields of each transaction, arranged from the highest (leftmost) to the lowest (rightmost) priority. Thus, the subfields are "bit-sliced" and loaded in the corresponding PLA stores. More details of the system organization are in [16].

The storage size of a particular PLA depends on the total number of transactions assigned to that PLA. Consider, for example, a sequencer configuration with one primary slice and two secondary slices, using the

following transactions:

```
M-CALL ADDR1,ADDR2,,,ADDR3,ADDR4
M-CALL ADDR5,ADDR6,ADDR7,ADDR8
M-CALL ,,ADDR9,ADDR10,ADDR11,ADDR12
```

The above transaction format is recognized by the microsequencer assembler to be discussed in the next section. The problem here is to partition the above code for assignment into the PLA structure of Fig. 7. A straightforward code segmentation would require three words assigned to each of the PLAs of Fig. 7. Some storage compaction can be achieved, however, by exploiting the empty fields, represented by commas in the above code. To illustrate the technique, suppose that A1, A2 and A3 represent the effective (input) address of the above M-CALL transactions. Then, these transactions can be "sliced" and compacted in PLAs 0, 1, 2 and 3 of Fig. 7, using the transaction field formatting of Section II, as follows:

```
PLA 0          A1// M-CALL
               A2// M-CALL
               A3// M-CALL

PLA-1          A1// ADDR1, ADDR2
               A2//ADDR5, ADDR6

PLA-2          A2// ADDR7, ADDR8
               A3// ADDR9, ADDR10

PLA-3          A1// ADDR3, ADDR4
               A3// ADDR11, ADDR12
```

As illustrated above, PLAs 1, 2 and 3 do not need storage in the designated addresses A3, A1 and A2, respectively. In fact, only two words are required for each of the 16-bit output PLAs, Fig. 7, and three words for the function opcode PLA, to store these transaction codes. This compaction technique is due to the associative mapping property of PLAs, and it is utilized in the PLA formatter tool (next section), resulting in substantial sequence storage reduction. Even better results may be obtained using a sophisticated PLA compaction algorithm, by column partitioning, in [18].

## V. MICROASSEMBLER AND FORMATTER TOOLS

The microcontrol architecture proposed is supported by several firmware design tools that have been developed and integrated into a software package, MMDS (Modular Microprogram Development System). MMDS is a general purpose tool aimed at the development of highly modular microprograms. A block diagram of MMDS is given in Fig. 8. It includes the following tools:

- a microsequencer and microcode assembler
- a microsequencer's PLA code formatter, and
- a functional-level modular microarchitecture simulator.

In this section we shall describe briefly the first two of the above tools; the simulator is discussed in the next section. More details are in [19] and in a user's manual [20].

## 5.1 Microsequencer and Microcode Assemblers

Microassemblers are programs that allow the encoding of a microprogram into source code and translation of this code into object code (bit-patterns) for loading into the control storage. The benefits of using microassemblers are similar to the ones accrued from using ordinary assemblers, and are well documented in [21]. The microcontrol scheme, due to the dual microprogram storage, requires separate code generation for microsequencing and microcoding. Although currently available microassemblers would be suitable for microcode generation, they could not be used to generate PLA sequencing code because they: 1) assign sequential addresses to consecutive microinstructions; 2) have fixed microword length for all microinstruction types; 3) do not support definitions of sequence type subfields or assignment of null values [20] to microorders.

Due to the above reasons, a microsequencer code assembler has been developed to convert control transactions, written for the microsequencer PLA in a specific format, into binary code. This software package contains two programs: 1) definition program, SDEF and 2) assembler program, SASM. SDEF allows the user to define formats of the firmware transactions in terms of subfield width, type and addressing mode. Several options have been provided for Hexadecimal, Decimal, Octal and Binary subfield values. The definition of symbolic constants is also allowed. SDEF creates a definition file and a listing file. The latter is produced for user's reference and it contains definition source and diagnostics. The definition file contains encoding information for the defined sequence transactions and symbolic constants. The assembler, SASM, is a two-pass program that transforms the definition source code into binary object code for the sequencer PLA.

The above software package was written in Pascal in a PDP11/60 minicomputer system. An example definition source is in Appendix I and more details are in [19].

A microcode assembler was generated by modifying the microsequencer code assembler. Two independent modules, MDEF and MASM, were generated from the preceding SDEF and SASM, respectively. The microinstruction definition module, MDEF, sets up the microcode word structure and mnemonic assignment for a given target machine. The microinstruction assembly program (MASM) translates the microcode source into bit patterns compatible with the target machine.

## 5.2 PLA Formatter Program

The purpose of the PLA formatter is to convert the, bit pattern, object module from the Microsequencer Code Assembler into a format suitable for downloading into the target PLA's. The formatter, specifically, partitions the object module into blocks of code depending on the PLA sizes. The program allows the user to specify the parameters of the target PLA (in terms of input variables, output function and product terms), the format of

the PLAs output code and the values of any don't cares in the bit pattern. In addition, the user may request a PLA Map and PLA code Output Listing.

A simplified diagram of the formatter is in Fig. 9. The input, i.e., the object module defines for each slice of the bit-slice microarchitecture, Fig. 7, the total number of words and the addressing range. In Fig. 7, the number of output functions is assumed to be four for slice-0 and sixteen for all other slices; however, the formatter has flexibility for other input/output arrangements. The object module is then partitioned by the formatter according to the user specified commands, into modules compatible with the specified target PLAs. The object module, for each slice, may be viewed as an array of PLAs, once the user has specified the number of product terms and output functions of the target PLAs. As mentioned in Section IV, the partition technique seeks to eliminate, as much as possible, the vacuous PLA fields in the microassembler generated object module to achieve compaction of the object module slices.

The formatter includes an interactive monitor, with a simple command menu, to provide user-oriented input. The monitor commands and other details are in [19]. An example of a PLA object module partition is in Appendix II.

## VI. MODULAR MICROARCHITECTURES SIMULATOR

The purpose of microprogram simulators is to simulate the data flow in a microprogram system. The importance and benefits of such tools are well documented in [22]. At present, there are two main methods for constructing a microprogram simulator. The first requires the user to fully define the actions of the machine in a procedural, register-transfer, language, e.g. ISPS [23] or N.mpc [24]. The second option is for the user to write a special machine-independent simulator, generally in a high-level language [25-26].

The method we adopted here is different from the above two techniques. The simulator is intended for target machines composed of commonly used bit-slice devices and functional/data modules (registers, shifters, multiplexers, memory, etc.). A library of simulation routines for these devices has been generated. This modular description promotes a top-down design approach and makes the design process into just a selection of standard cells. The target machine is directly implemented by calls to these routines. This give the simulator a reasonable amount of execution speed compared to the register-transfer language method, while providing reasonable flexibility in defining different target machines. The organization of the Modular Microarchitectures Simulator is shown in Fig. 10. It is subdivided into five independent modules: SAI Assembler, Interactive Monitor, Simulation Monitor, Supervisor and Microsequencer.

The SAI (Storage Allocation and Initialization) Assembler is a one pass assembler which reads and analyzes the user supplied Storage Allocation and Initialization file and generated a table of the user defined symbols and an Output Listing file for user's reference. The Interactive Monitor provides a User-Simulator communication. An easy to use command language for microprogram testing and debugging has been provided [19].

The Simulation Monitor controls the microprogram execution during simulation. The control flow is shown in Fig. 11 . In the beginning, the simulator's microprogram counter is initialized, as to the number of microcycles to be simulated; an output trace file is also initialized. Thereafter, the simulator enters an execution loop. For each microcycle, the microsequencing and microcode information is fetched from the corresponding (simulated) sequence and control stores and an entry is made in the trace file, if requested, as to the counter value and transaction being executed. After the specified number of microcycles, the simulator exits the loop or, it exits under any special conditions such as out-of-range address, break point address, etc.

As shown in Fig. 11, the simulator may be operated in the mapped, pipelined or non-pipelined modes. The mapped mode is used when simulating an external sequencer, e.g., AM2910, in the target machine. When using the internal PLA sequencer, the Simulator may be operated in either the pipelined or non-pipelined mode. In the pipelined mode, a parallel execution of the Supervisor and Microsequencer modules, Fig. 10, is performed. By contrast, a serial execution of these modules is simulated in the non-pipelined mode. At any rate, the Simulator monitor coordinates the status and address processing by the Supervisor and Microsequencer, respectively, to be discussed next.

The Supervisor is a program module implementing in its program structure the target machine architecture by calls to a cell library of simulation routines, along with timing information. Different target machines require changes only in the Supervisor structure. The Supervisor essentially performs two tasks. First, it maps the microinstruction fields controlling each device into the corresponding routines; second, it executes the microoperations as calls to these routines. The source code required for the Supervisor is small and a target machine is easily defined in its program structure thus making this technique very flexible.

The two tasks of microcontrol, i.e. microcoding and microsequencing are performed by the Supervisor and Microsequencer modules, respectively, in the non-mapped simulation mode (Fig. 11). The Supervisor requires a microinstruction word as input from the Simulator and returns the status along with other relevant information (mapped address, loop count) to the Simulator for microsequencing. In the mapped mode, the supervisor also does the microsequencing, returning a mapping address after each microinstruction execution.

The Microsequencer is a software model of the hardware control scheme discussed earlier. It has been incorporated into the Simulator to encourage the development of modular microprograms and relieve the designer from the task of sequencing in the initial design phase. All stacks in the address processor of Fig. 6 are available to the user for modification/examination to aid debugging. Depending on the sequencing transaction being executed, the Supervisor supplies the required external inputs to the address processor (external status signals, mapping address, loop count).

# VII. TEST EXAMPLE: BINARY SEARCH TREE MIGRATION

A binary search tree (BST) algorithm has been selected as a test example to demonstrate the proposed control architecture, the usefulness of the firmware sequencing constructs and the usage of the microprogram development system. The example chosen is a suitable, non-trivial, candidate for firmware migration. For this purpose we use a versatile target machine architecture based on the AM2910's and controlled by the microsequencer scheme. The details are discussed next.

## 7.1 Algorithm

Binary search trees are often used to build symbol tables in loaders, assemblers, compilers or any keyword driven translator [27]. By definition, a BST is a binary tree; if not empty, the BST node identifiers satisfy the following: 1) all identifiers in the left (or right) subtree of BST are less (or greater) than, numerically or alphabetically, the identifier in the root node of BST; 2) the left and right subtrees of BST are also binary search trees [27].

The structure of a node of BST, illustrated in Fig. 12(a), consists of LLINK (left link), RLINK (right link), IDENT (identifier) and data fields. The latter may be of variable or fixed length. For convenience, a header node is also included in the BST structure such that the actual BST forms the left subtree of the header; the other header fields are empty.

The BST algorithm is given in Fig. 12(b). The notation used follows from the following formulation: search BST with header H for node C such that IDENT(C) = IDENT(E). Set usf(user flag) if C is found, else insert node E at the appropriate point in the tree.

## 7.2 Target Machine Structure

The target machine, shown in Fig. 13(a), for implementing the BST test example consists of four main parts: the data path (processor), the controller, the pipeline register and the status register. The controller essentially comprises one-slice configuration of the microcontrol scheme discussed earlier (Fig. 7). An instruction-data based pipelined scheme is used offering significant improvement in speed. The 25-bit pipeline register carries the (current) microcode word that controls the data path. The bit assignment is shown in Fig. 13(b). The status register output is connected to the three least significant bits of the external status bus in the address processor of Fig. 6(a).

The data path structure of the target machine is shown in Fig. 14. The control part of the structure is a set of four AM2910 ALU slices. The address and data bus are sixteen bits wide. The main memory is assumed to be a 16 bits x 64 words RAM. The MBRIN, MBROUT and MAR are all 16 bit registers used for memory read/write operations. During a memory read, the contents of the location addressed by MAR is read into the MBRIN. During a memory write, the contents of MBROUT is written into the location addressed by MAR. The status register, shown again in Fig. 14, is three bits wide and holds the USF, the Z (zero) and N (sign) outputs from AM2910. The control inputs to these elements are as shown.

The timing of the target machine is controlled by a system clock. Timing assumptions and other timing details are in [19].

### 7.3 Firmware Description

A top-down design approach is used in the firmware design of this example. The control flow, shown in Fig. 15, follows directly from the previous BST algorithmic description. The first four microinstructions perform the initialization. The while-do loop in the algorithm is replaced by the SLOOP sequencing transaction of Table I. For modularity, the operations performed within the while-do loop are replaced by a microprogram module, SEARCH. Also, another module, INSERT, has been defined, which contains the microoperations required for inserting the element into the tree. The conditional call to INSERT is achieved by the SCALL transaction. The microoperation flow chart for the SEARCH and INSERT modules is shown in Fig. 16(a) and (b), respectively. The case structure in the algorithm is formed by the MJUMP transaction.

This example clearly demonstrates the power of the sequencing constructs provided by the microcontrol scheme and also the structured approach for firmware design. All the I/O files for simulating this microprogram on the microprogram development system are in Appendix A of [19]. A listing of the User-Simulator interaction for inserting nodes E and F into a BST is in Appendix B of [19].

## VIII. SIGNIFICANT RESEARCH RESULTS

We reported previously on the design of a PLA-based microcontrol scheme supported by structured firmware primitives that allow complex control sequencing. We also reported on MDSS, a set of microprogram development tools constructed for this purpose. In the course of this work our original research objectives broadened. It did not appear sufficient just to build a microcontroller, however powerful its sequencing capability might be. What was also important was the capability of "good" mapping of complex functions into the serquencing constructs of the microcontroller. More specifically, the following requirements are also important: 1) how the microsequencing scheme could be used to realize complex software functions in firmware, i.e., firmware migration. 2) implementation of such functions in microcoded silicon structures such as VLSI PLAs.

Further research was pursued to establish the feasibility of the above objectives. This research, still underway, has three main thrusts.

1. Function firmware migration

2. VLSI microcode implementation

3. Microarchitecture definition via hardware design language

We have worked on all the above thrust areas but further work is still needed to produce an integrated system approach. We will give here a summarized report on the most important results we have obtained in our investigation of these areas. More details are in several references

published by our group [18,28,29].

8.1 Firmaware Migration

Migration of frequently-used software into firmware is a well-known technique for improving the system performance. However, firmware migration has been influenced by VLSI technology due to the capability to embed in silicon not just "traditional" microprograms but also complicated software functions such as parsers or operating system primitives. In general, such function have complex logical structure. Thus, cost-effective migration requires modular microprogram structures with powerful sequencing capability. The basic objective of our work in this area is to explore an automated software-to-firmware migration technique based on PLA-oriented microcontrol architectures reported earlier. The approach is briefly described next.

The basic idea is to extract the sequencing structure via compilation techniques. The selected function for migration is processed in several phases by the automatic migrator, Fig. 17(a). The end result is microsequencing code describing the sequencing structure of the function. The code consists of the sequencing constructs, reported earlier, particularly calls to microcode modules. The latter comprise the structured firmware implementation of an instruction set on a base machine on which the migrating function is tested. If this firmware code is not available, it may be produced by the microcode emulator, reported earlier, to be executed on the base machine. The various phases of the proposed migration scheme are shown in Fig. 17(b) and are discussed in our publications.

For the base machine, a bit-slice architecture is used to provide flexibility, expandability and modularity. The entire base machine structure has been simulated on a PDP11/60 and is supported by firmware tools, developed earlier. The instruction set of the 11/60 is emulated on the base machine in a modular fashion, i.e., each 11/60 instruction has a microcode "module" resident in the (simulated) micromemory of the base machine. Thus, migration is performed by sequence calls to microcode modules interpreting the base machine on which the function is tested. Again, these sequence calls express the sequencing structure of a function, and are generated by the migrator. A case study of the migration sheme has been detailed in section VII.

Among the advantages of the scheme is that it does not require familiarity with machine details for the user. Further, the control storage is significantly reduced as migration is implementated through sequence calls. The scheme utilizes the microcode existing in the processor avoiding microcode repetition. Some experimental results with five software functions as migration candidates are very encouraging demonstrating an improvement factor of about 5. These results are discussed in the previous reference. However, additional work is needed to establish this approach to function migration.

8.2 Microcode in VLSI structures

In the second research direction we obtained significant results by a new (chip) area reduction technique suitable for PLA microcode. This compaction is very much related to the migration technique if one wants to implement a function into VLSI code rather than the conventional firmware. The fundamental building blocks of VLSI are PLAs, so far. Previous PLA compaction techniques view the PLA as a random Boolean matrix. However, in microcoded PLAs, the information is regularly organized into fields, including a large proportion of empty or don't care fields. Our approach is to eliminate, as much as possible, the empty fields by partitioning the PLA, by columns, into a number of smaller, but denser, arrays which require less overall area. An important contribution of this work is an area reduction algorithm based on a breadth-first graph searching approach. The experimental results are very encouraging and are detailed in our publications.

We recognized that the regularity of a microcode matrix resembles other data tables organized in regular information fields. These data structures appear quite frequently in software design of parsers and hash tables and they are candidates for silicon compilation. Thus we consider the more general problem of designing data tables in VLSI microcode. The goal is to compose a Design Automation system for the PLA implementation of such tables in VLSI. A related objective is to integrate this DA system to other existing tools in our research environment at various design levels, i.e., the architecture level (hardware design language MDSL), the firmware level (microprogram development system MDS), and the layout level that includes layout packages, PLA generators, cell libraries and the like. A fundamental issue involved here is compaction. In our approach, we investigated a new compaction technique based on partition and fusion. Some details of our approach follow.

Due to their regular construction, the PLAs are now standard components of VLSI chips and, consequently, PLA compaction is quite significant in in VLSI design. There are basically three types of PLA optimization techniques in the literature: PLA minimization, PLA folding and PLA partitioning. The main characteristic of the above techniques is that they view the PLA as a random logic function. However, there are many applications where PLAs contain more formatted or structured information. For example, when a PLA is used as a microcode store, the structure of the stored data tends to be somewhat regular. This regularity and, at the same time, sparsity of information also appears in several other data tables which are important in software such as hash tables, parsers, symbol tables, etc. We believe that an important ingredient for the migration of software in VLSI will be the efficient implementation of data tables by PLA structures.

In this work, we propose a new PLA compaction technique which exploits the regularity of the information embedded in the PLAs. Our approach involves first PLA partitioning and, second, PLA fusion. Partition is performed by column splitting of the data table on the basis of a heuristic search technique using a directed graph. Fusion is performed using an encoding scheme to map the indexes of common data fields in the partitioned PLAs into distinct fields referenced by fused index block code in the search (AND) arrays.

Although our partition technique applies to any data table, it is clearly superior to the other PLA partition schemes when it is employed on structured tables. The PLA fusion technique is unique, to the best of our knowledge. To support these claims we organized an experiment, based on a software implementation of our technique to determine statistics of PLA compaction by partition and fusion for randomly generated but structured data tables. Specifically, we studied the chip area compaction with respect to the original (unreduced) PLA size for several samples of various data tables, implemented by the proposed technique. There were about 1000 data tables processed resulting in 85% successes, i.e., reduced tables that did not require more address bits than without compaction. In fact in some cases we ended up with reduced tables that actually required fewer address bits than the original tables. The results are concisely depicted in Fig. 18. The main observations are:

1) Larger tables are more suitable for the proposed compaction method.

2) The savings in chip area are larger for greater sparcity (lower density) tables.

3) The number of table fields as well as the width of the fields did not appear to have a marked effect on the area saved.

### 8.3 Hardware Design Language MDSL

In our approach, the overall problem associated with a design automation system may be divided into three major steps: First, to devise a method for expressing and collecting the structural and behavioral information of a target machine architecture. A hardware description language that fulfills this task has already been constructed. Second, to establish a software translation process that will convert input descriptions based on the language constructs into target microcode. Finally, one must introduce a methodology for extracting the information needed for target machine simulation. Some details of our approach follow. More details are in our publications on the MDSS system.

The language for the description of microarchitecture MDSL (Microcode Development and Simulation Language) is a language suitable for the description of microprogrammed microprocessors at the register-transfer level. It contains the facilities to describe the structural and behavioral information, simultaneously, for the complete specification of a microprogrammed The structural information is defined in the <u>structure section</u> which is divided into a number of subsections. An important subsection is the <u>element</u>, distinguished into the storage, the link, the input/output and the functional element types. The storage definitions are descriptions of registers, subregisters, and memory components of the system, with the link element definitions containing appropriate information regarding their control point locations. Similarly, the functional element definitions contain the information concerning the register-transfers and their control requirements. There are three additional subsection types to be defined in the structure section: the sequencing scheme, the special function routines and the control format.

The behavioral information is defined in the behavior section which contains the instruction-set definition of the register-transfer microoperations.

The control requirements are defined explicitly in the functional element definitions, and implicitly in the link element and instruction-set microoperations definition. The control word format declared in the format subsection can be used to map the control requirements of the instruction-set microoperations into microcode. The concurrency of the microoperations can be checked to determine the control word combinations.

The MDSL (Microcode Development and Simulation Language) is an efficient tool to develop microcode and simulate the hardware operations of microprogramed processors. A software translator for MDSL has been developed for this purpose. An important aspect of this translator is that, in lieu of machine object code, it generates data structures by parsing MDSL input descriptions. Those structures are used to generate C language programs which are compiled for the simulation and the microcode generation.

In the process of generating efficient microcode, the element description can be translated into a microcode template table. Each template contains information concerning the source and destination storage devices, and the control requirements. This template table can be used to map the behavioral description statements into microcode, provided that the behavioral information can be translated into data structures resembling the template. The microcode generator contains a facility for local optimization into horizontal microcode format.

## IX. SUMMARY AND CONCLUSION

We have presented in this report two PLA based microcontroller architectures which have the capability of complex sequencing such as multiway branching, microsubroutines, nested microlooping, and the like. The basic components of the first microcontroller are a PLA sequencer store, an address generating processor and a microcode store composed of PLAs or ROMs. Microsequencing and microcoding are thus separated and embedded in the corresponding stores. In addition to increasing the sequencing capability, this approach reduces the sequence storage as implicitly generated sequencing information need not be store in the PLA. A bit-slice approach was taken in the second microcontroller consisting of parallel PLA sequencer slices along with corresponding address processing elements and a microcode store. This structure avoids pin constraints, allows expandability and results in further compaction of the sequence storage.

The separation of microsequencing and microcoding enhances the modularity of the schemes. Thus, the addition of new modules of microcode in ROM simply requires the insertion, in the PLAs, of sequencing information about the entry and exit points of the modules. The regularity of the structure and of its components constitute a favorable environment for LSI/VLSI implementation. Using VLSI/CAD tools, already available, it is quite feasible to compact an address processor slice together with its PLA slice into a single VLSI chip. This may further solve the pin difficulties

and improve the chip area efficiency. A VLSI design of the single-slice microcontroller, beyond the scope of this report, is reported in [28].

The proposed microarchitecture realizes our basic objective, the structured firmware design and implementation of modular microprogramming. Modularity is an important prerequisite for the migration of complex software, e.g., operating systems, into firmware for reasons of speed, reliability and stability. To facilitate modular microprogramming, we have used at the microlevel, the basic control primitives of structured programming (if-then-else, while-do, case, etc.). Complex constructs have also been developed to perform compound loop sequencing. The PLA based architectures realize much more powerful sequencing functions than the existing microsequencers such as the AM2910. Moreover, the proposed constructs are more suitable for modular microprogramming than the rather unstructured primitives of the commercial microsequencers.

The development and debugging of microprogram is a task of high significance and complexity and requires suitable firmware tools. Several such tools have been developed and integrated into a software package called MMDS (Modular Microprogram Development System). It includes the following: a microsequencer and microcode assembler, a PLA code formatter, and a modular microarchitectures simulator. As a test example, a binary search tree algorithm was coded in the sequencing constructs by means of MMDS for a simple target machine.

A research initiative was undertaken to investigate the general problem of function migration in firmware and the feasibility of implementation of such migrations in VLSI microcode. An automatic migrator based on the PDP11/60 instruction set interpretation was constructed and tested. A method for implementing the migrated function in compacted PLA microcode was introduced. We also developed a hardware design language approach for structural and behavioral definition of architectures and optimized microcode generation. The language tools will be important in the continuation of our research effort, the design of a retargetable migrator of complex functions into microcoded VLSI microarchitectures.

## REFERENCES

1. M. V. Wilkes, "The best way to design an automatic calculating machine," Manchester University Computer Inaugural Conference, pp. 16-18, 1951.

2. S. G. Tucker, "Microprogram Control for System/360," IBM Systems Journal, Vol. 6, pp. 222-241, 1967.

3. M. J. Flynn, "Interpretation, microprogramming and control of a computer," in Introduction to Computer Architecture (H. S. Stone ed.), Palo Alto, CA: Science Research Associates, 1980.

4. A. B. Salisbury, Microprogrammable Computer Architectures, New York: Elsevier-North Holland, 1976.

5. J. A. Stankovic, "The types and interactions of vertical migrations of

functions in a multi-level interpretive system," IEEE Trans. on Computers, Vol. C-30, No. 7, pp. 505-513, July 1981.

6. J. Stockenburg and J Van Dam, "Vertical migration for performance enhancement in layered hardware/firmware/software systems," IEEE Computer Magazine, Vol. 2, No. 5, pp. 35-50, May 1978.

7. Advanced Micro Devices, Bipolar Microprocessor Logic and Interface Data Book, Sunnyvale, CA: Advanced Micro Devices, 1982.

8. C. Mead and Conway, Introduction to VLSI Systems, Reading, MA: Addison-Wesley, 1980.

9. J. A. Fisher, "2-way jump instruction hardware and an effective instruction binding method," in Micro-13, 13th Annual IEEE Microprogramming workshop, New York: IEEE, pp. 64-75, Oct. 1980.

10. A. W. Nagle, R. Cloutier, and A. C. Parker, "Synthesis of hardware for the control of digital systems," IEEE Transactions on CAD of Integrated Circuits and Systems, Vol. CAD-1, No. 4, pp. 201-212, Oct. 1982.

11. R. W. Marczynski and M. S. Turdruj, "Microprogrammed control units towards modularity in microprogramming," Proc. Second Symp. on Micro-Architecture, Euromicro, 1976, North-Holland Publishing Company, pp. 173-181.

12. M. S. Tudruj and R. F. Gajda, "The modular firmware architecture through the stack/register based address modification," in Firmware, Microprogramming and Reconstructurable Hardware, North-Holland, 1980.

13. M. Andrews, Principles of Firmware Engineering in Microprogram Control, Rockville, Md.: Computer Science Press, 1980.

14. T. Kohonen, Content-Addressable Memories, New York: Springer-Verlag, 1980.

15. C. Papachristou and S. B. Gambhir, " A microsequencer architecture with firmware support for modular microprogramming," Micro-15, 15th Annual IEEE Microprogramming Workshop, New York: IEEE, pp. 105-113, Oct. 1982.

16. C. Papachristou and S. B. Gambhir, "A bit-slice microcontrol architecture for structure firmware designs," IEEE International Workshop on Computer Systems Organization, New York: IEEE, pp. 154-163, March 1983.

17. J. Mick and J. Brick, Bit-Slice Microprocessor Design, New York: McGraw-Hill, 1980.

18. C. Papachristou and J. Reuter, "Microprogramming and area reduction techniques for PLA microcode", 17th Annual IEEE Microprogramming Workshop, New York: IEEE, pp. 86-94, Nov. 1984.

19. S. B. Gambhir, Microcontrol Schemes for Structured Firmware Designs and a Modular Microprogram Development System, M.S. thesis, University of Cincinnati, 1983.

20. S. Gambhir and C. Papachristou, Microsequencer and Microcode Assembler Manual, University of Cincinnati, College of Engineering, 1984.

21. G. Myers, Digital System Design with LSI Bit-Slice Logic, New York: John Wiley, 1980.

22. D. Lewin, Computer-Aided Design of Digital Systems, New York: Crane Russak, 1977.

23. M. Barbacci and D. Siewiorek, The Design and Analysis of Instruction Set Processors, New York: McGraw-Hill, 1982.

24. C. Rose, G. Ordy and F. Parke, "N.mPc: a retrospective," 20th Design Automation Conference, New York: IEEE, pp. 497-505, June 1983.

25. P. Corcoran, "Simulator generator system," IEE Proc., Vol. 128, Pt. E, No. 2, pp. 61-63, March 1981.

26. M. Mezzalama and P. Prinetto, "Design and Implementation of a Flexible and Interactive Microprogram Simulator," Micro-12, 12th Annual IEEE Microprogramming Workshop, New York: IEEE, pp. 43-48, 1979.

27. E. Horowitz and S. Sahni, Fundamentals of Data Structures, Rockville, Md: Computer Science Press, 1976.

28. C. Papachristou, R. Rizwan and S. B. Gambhir, "VLSI design of a PLA-based microcontrol scheme," IEEE Internat. Conference on Computer Design: VLSI in Computers (ICCD-84), New York: IEEE, pp. 771-777, Oct. 1984.

29. C.Papachristou and J.-P.C. Hwang, "Computer-aided design of digital systems: Language, data structures and simulation," invited contribution to Advances in Management and Information Systems, Vol. II: Languages for Automation (S.-K. Cang ed.), New York: Plenum Publishing Corp., pp. 465-484, July 1985.

## PUBLICATIONS RELATED TO RESEARCH PROJECT


1. "Generation and implementation of state machine controllers: a VLSI approach," Microprocessing and Microprogramming, (North-Holland), Vol. 16, No.3, October 1985, (C. Papachristou and D. Cornett).

2. "Computer-aided design of digital systems: Language, data structures and simulation," invited contribution to Advances in Management and Information Systems, Vol. II: Languages for Automation (S.-K. Cang ed.), New York: Plenum Publishing Corp., pp. 465-484, July 1985. (C. Papachristou and J.P.-C. Hwang).

3. "A PLA microcontroller using horizontal firmware, Microprocesssing and Microprogramming (North-Holland), Vol. 14, No. 3-4, pp. 223-230, November 1984 (C.Papachristou).

4. "Microcode development for microprogrammed processors, 18th IEEE-ACM Microprogramming Workshop, New York: IEEE, December 1985, pp. (J.P.-C. Hwang, C.Papachristou and D. Cornett).

5. "PLA compaction by partition and fusion," IEEE Internat. Conf. on Computer-Aided Design, (IC-CAD), New York: IEEE, pp. 166-168, November 1985 (C.Papachristou).

6. "Microassembly and area reduction techniques for PLA microcode," 17th IEEE-ACM Microprogramming Workshop, New York: IEEE, pp. 86-94, November 1984 (C.Pachristou and J.Reuter).

7. "An automatic migration scheme based on modular microcode and structured firmware sequencing," 17th IEEE-ACM Microprogramming Workshop, New York: IEEE, pp. 155-164, November 1984 (C.Papachristou, R. Immaneni and D.Sarma).

8. "MMPS: Modular microprogramming compiler, simulator and CAD tool," 1984 IEEE Workshop on Languages for Automaton, New York: IEEE, pp. 101-106, November 1984 (C.Papachristou and E.Melton).

9. "VLSI design of a PLA based microcontrol scheme," IEEE International Conference on Computer Design: VLSI in Computers (ICCD-84), New York: IEEE, pp. 771-777, October 1984 (C.Papachristou, R.Rashid and S. Gambhir).

10. "A language for digital system specification and design," 1983 IEEE Workshop on Languages for Automation, New York: IEEE, pp. 229-237, November 1983 (C.Papachristou and P.-C.Hwang).

11. "A bit-slice microcontrol architecture for structured firmware designs," IEEE International Workshop on Computer System Organization, New York: IEEE, pp. 154-163, March 1983 (C.Papachristou and S.Gambhir).

12. "A microsequencer architecture with firmware support for modular microprogramming," <u>15th IEEE-ACM Microprogramming Workshop</u>, New York: IEEE, pp. 105-113, October 1982 (C.Papachristou and S.Gambhir).

13. "A CAD system for implementing state machine controllers," <u>11th Euromicro Symposium on Microprocessing and Microprogramming</u>, September 1985 (C.Papachristou).

14. "Modular microcontrol development system using PLA firmware," <u>10th Euromicro Symposium on Microprocessing and Microprogramming</u>, August 1984 (C.Papachristou).

PERSONNEL SUPPORTED BY THE PROJECT AND

DEGREES AWARDED DURING THE PROJECT PERIOD

| | | |
|---|---|---|
| 1. | Evelyn A. Melton | M.S. degree, June 1983 |
| 2. | Satnamsingh B. Gambhir | M.S. degree, August 1983 |
| 3. | Rizwan Rashid | M.S. degree, December 1983 |
| 4. | James Reuter | M.S. degree, June 1984 |
| 5. | Jerry P.-C. Hwang | Ph.D. degree, August 1984 |
| 6. | Rao Immaneni | M.S. degree, December 1984 |
| 7. | Anil Pandya | M.S. degree, May 1986 (expected) |
| 8. | Cris Koutsygeras | Ph.D. degree, May 1987 (expected) |

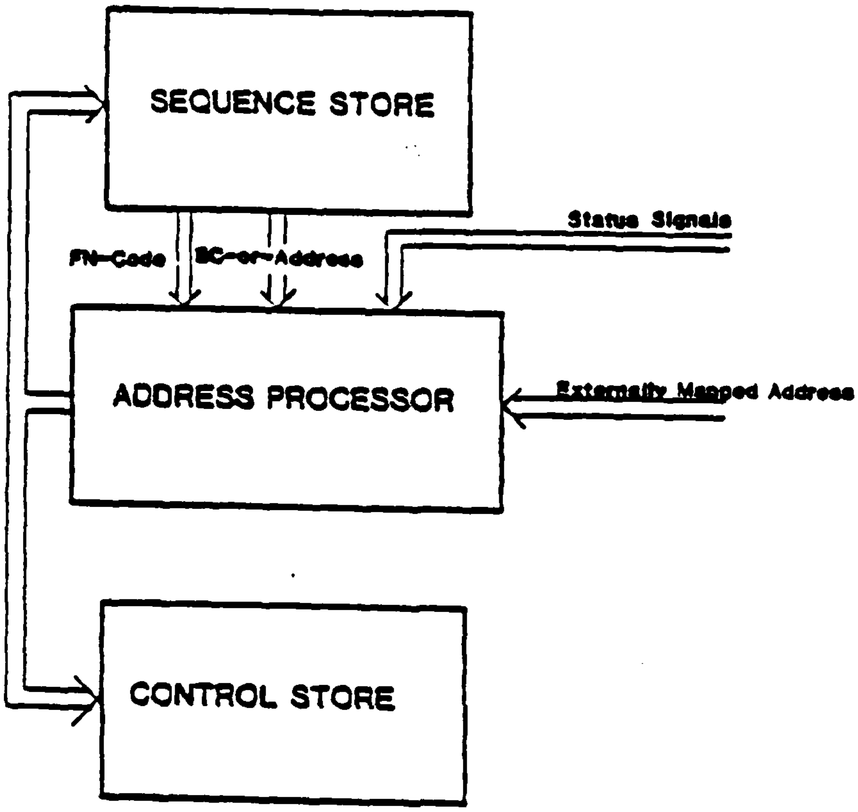



Fig. 1: Microsequencer Architecture

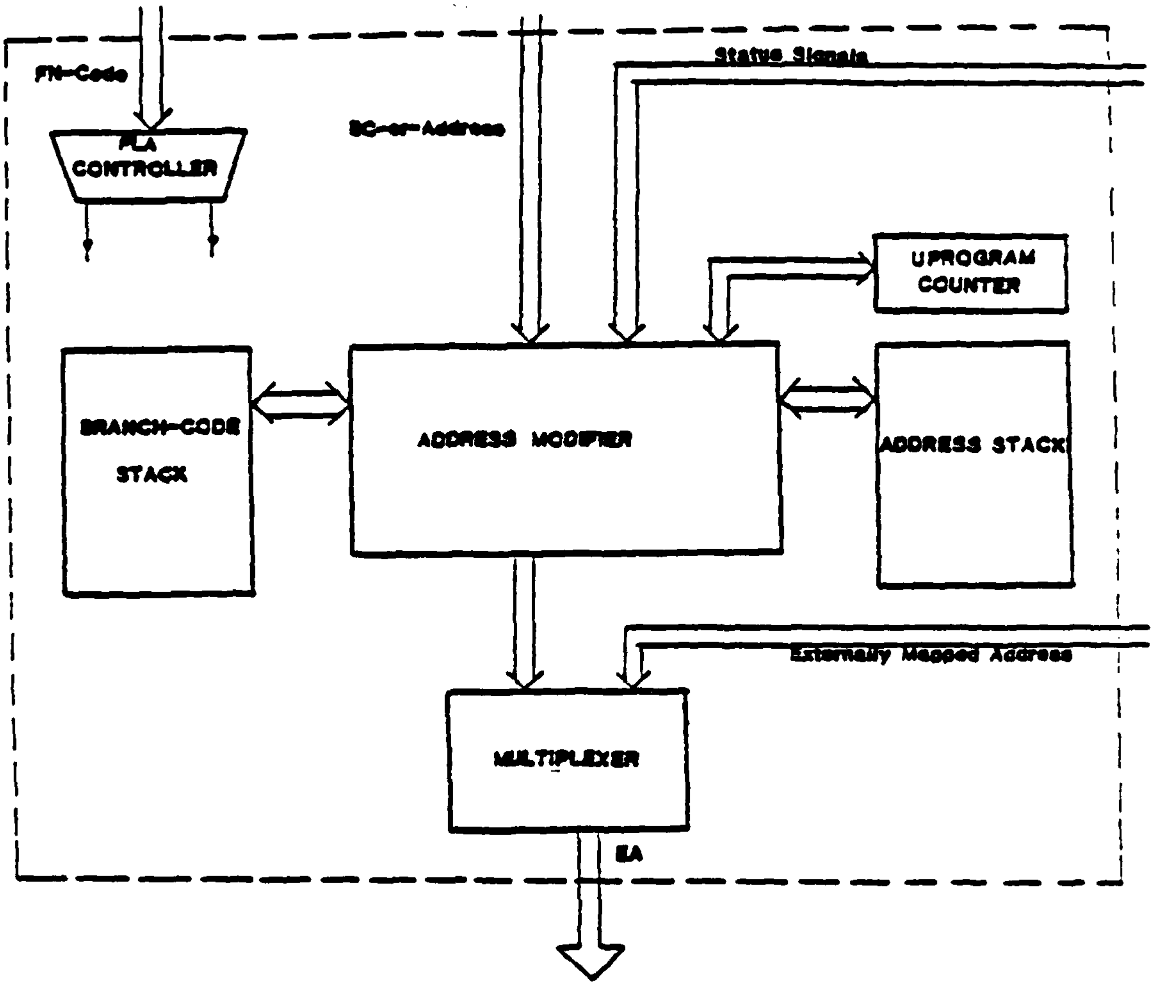



Fig. 2 : Address Processor

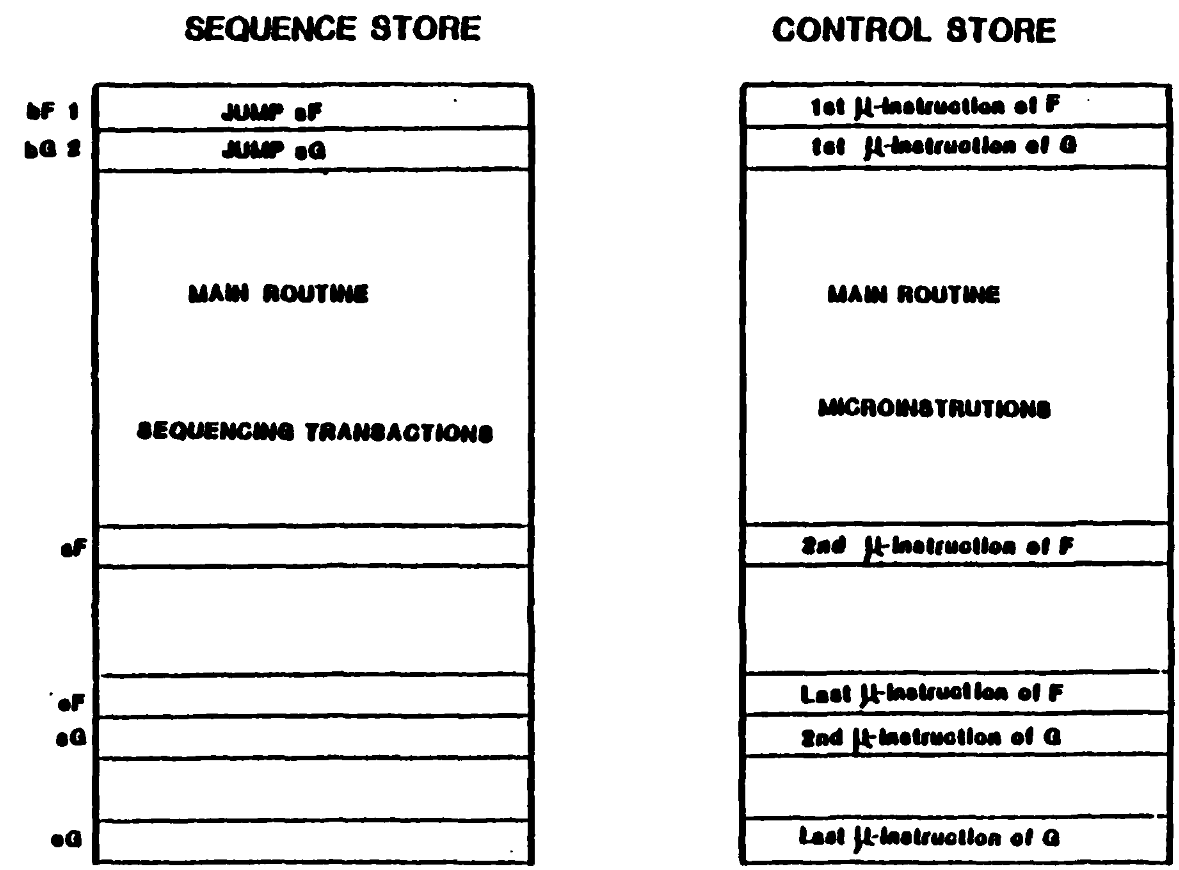
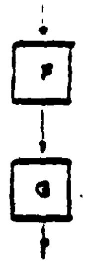


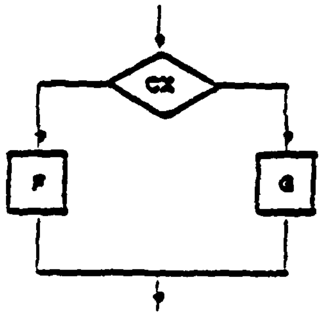

Fig. 3 : Organization of Microprogram Memories

(a)



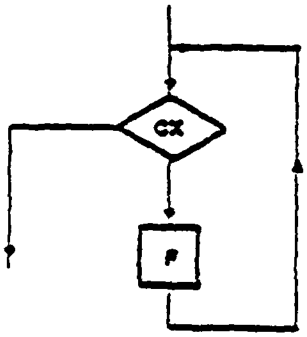

```
(a) Sequential Construct: F then G
/AX//CALL/NX=bF/ ;Unconditional call
/AY=AX+1/CALL/NY=bG/
/AZ=AY+1/____
```

(b)



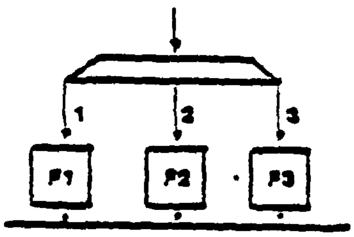

```
(b) Conditional Construct: If CX then F else G
/AX//SCALL/CX,bF/   ;binary conditional call
/AY=AX+1/CALL/bG/   ;unconditional call
/AZ=AY+1/___
```

(c)



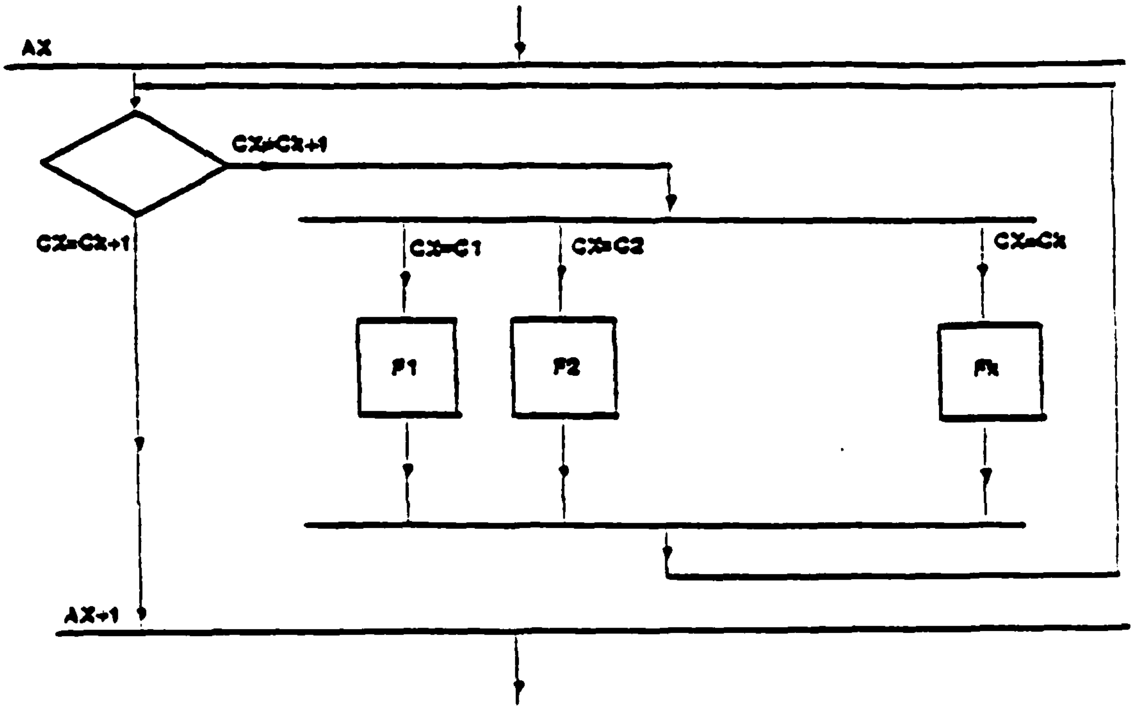

```
(c) Loop Construct: While-do
/AX/SLOOP/CX,bF/  ;binary conditional loop
/AY=AX+1/___
```

(d)




```
(d) Multiple conditional: If Case k then Fk
/AY=AX-1//SBC/C1,C2,....,Ck/ ;store branch code
/AX/MCALL/bF1,bF2,....,bFk/  ;multiway modular call

                              block F:
                              /AbF//JUMP/NAbF=eF/
                              ------   -------
                              /AsF//__/____
                              -----    -------
                              /AeF//RTN/
```

*Fig. 4: Formulation of firmware transactions*
*(a) sequential (b) conditional*
*(c) iterative (d) multiway (case-of)*
*The structure of all blocks is as in F.*




```
/AY=AX-1//SBC/C1,C2,....,Ck/
/AX//MLOOP/bF1,bF2,....,bFk/  ;modular loop
/AZ=AX+1//________
```

*Fig. 5: Modular looping in firmware (the structure of all blocks is as in Fig. 4).*

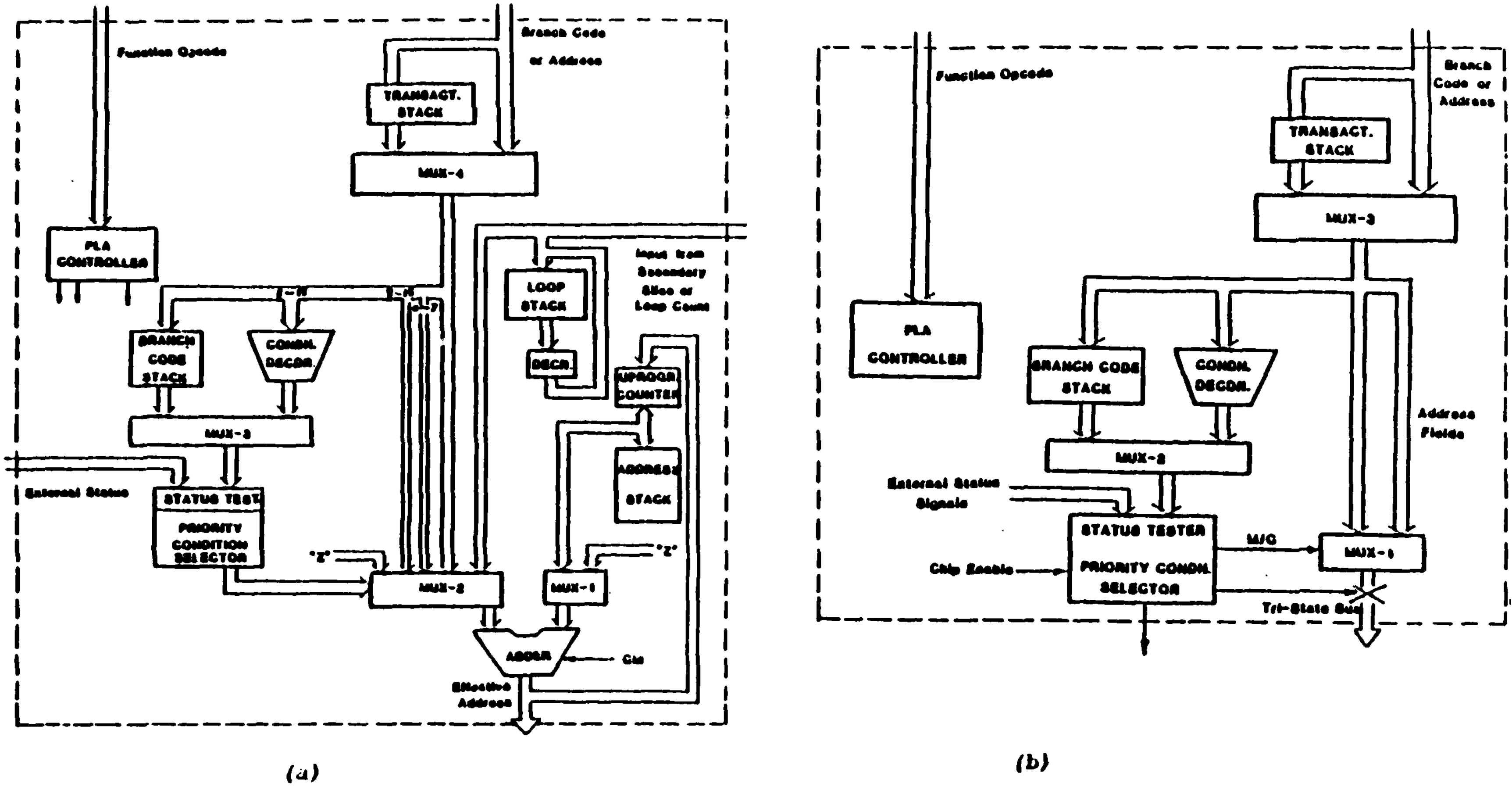



*Fig. 6: Address processor slices; (a) Primary slice, (b) Secondary slice*

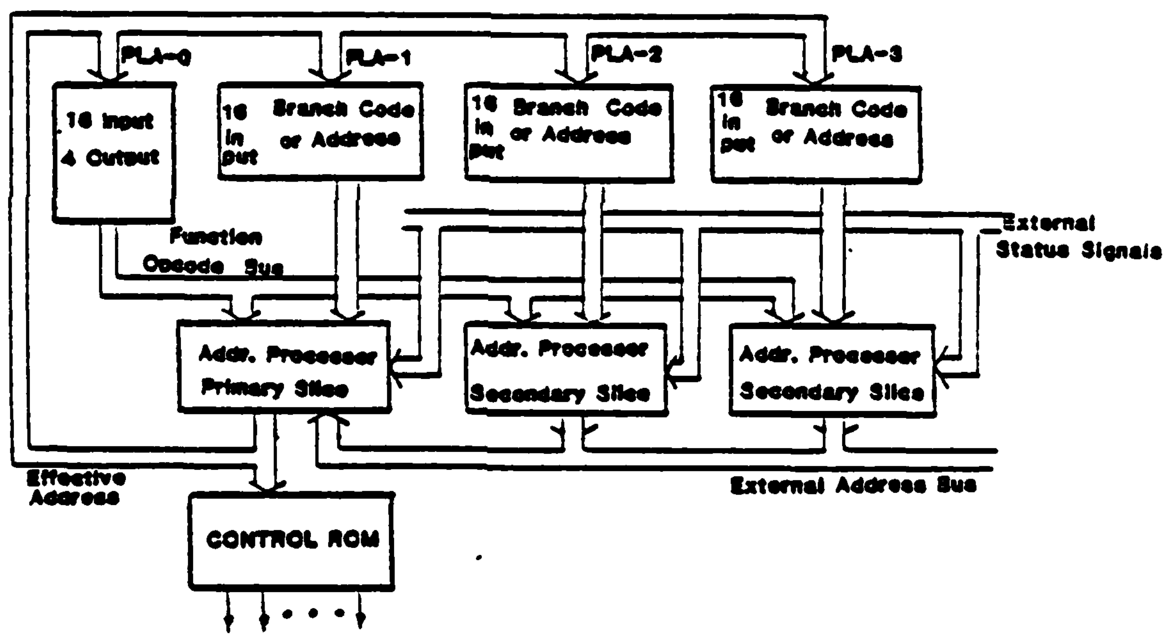



Fig. 7 : Bit-Slice Microcontroller Organization

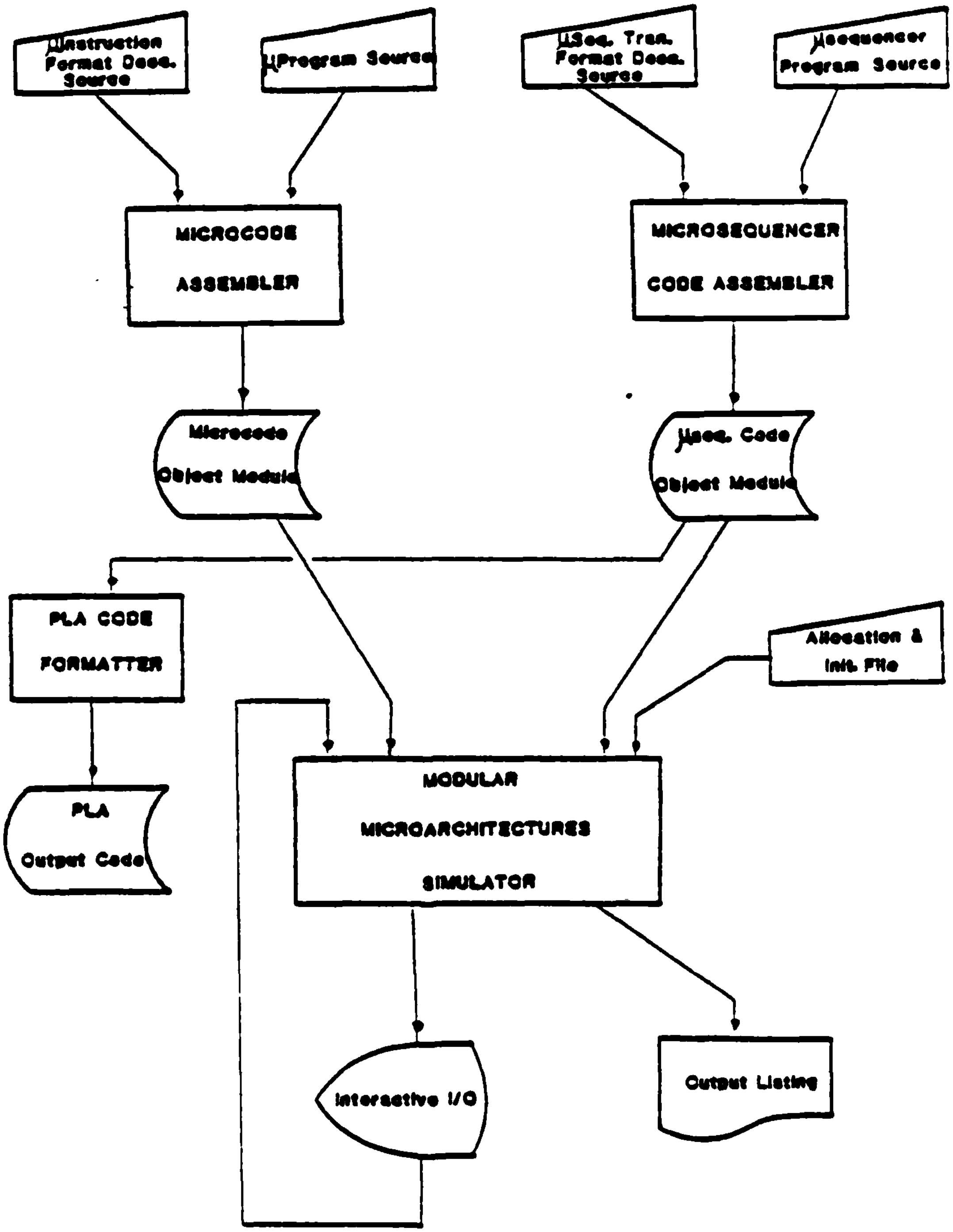



Fig. 8 : Modular Microprogram Development System

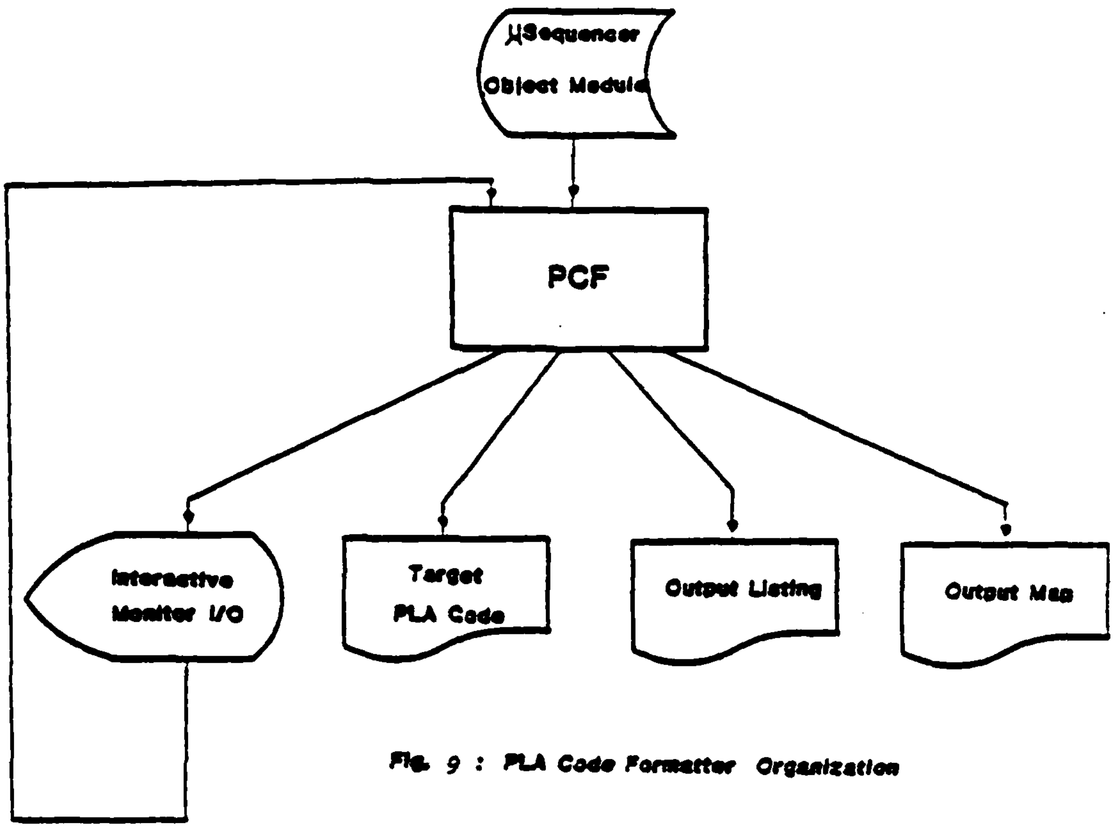



Fig. 9 : PLA Code Formatter Organization

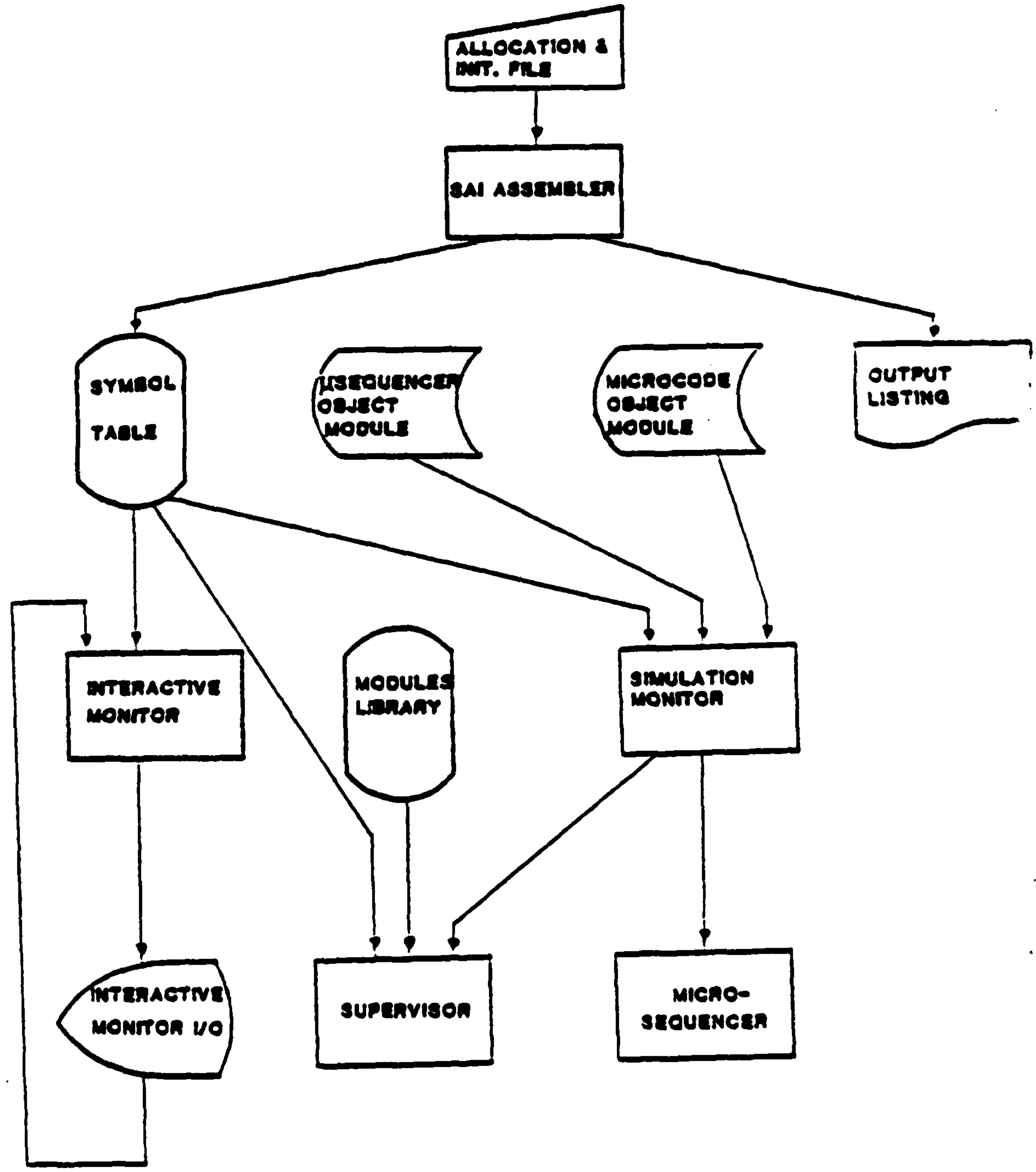



Fig. 10: Modular Microarchitectures Simulator

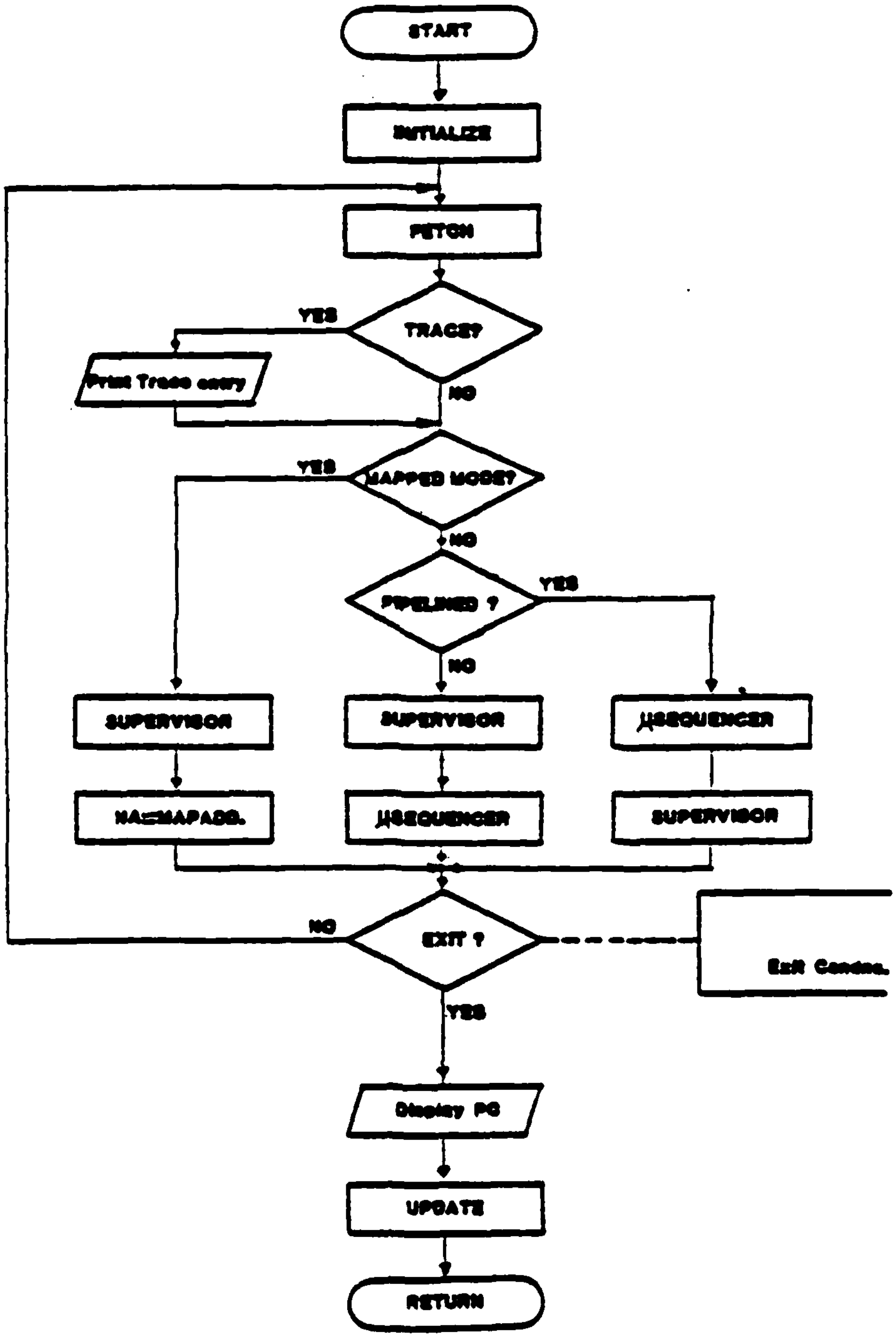


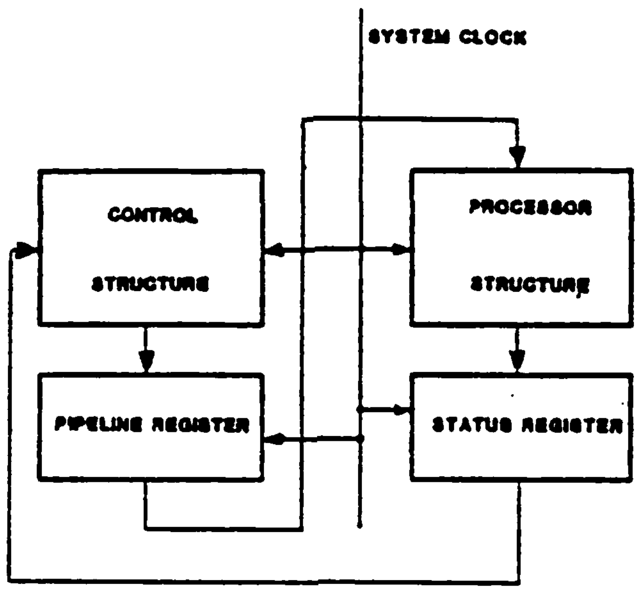

Fig. 11: Simulation Monitor routine

| LLINK | RLINK | IDENT | DATA |
|---|---|---|---|

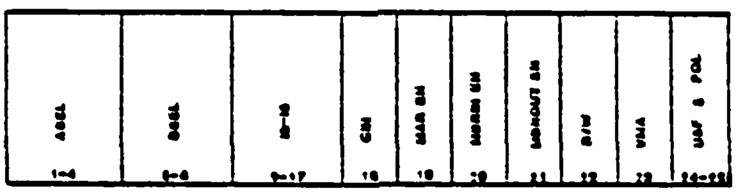
Fig. 12a: Structure of a Node

*Fig. 12b:* The algorithm for binary tree search and insertion is given below in a procedural language.

```
PROCEDURE BISRCH(VAR H,E:PTR;VAR USF:BOOLEAN);
VAR C,L:PTR;
  {C WILL POINT TO THE NODE BEING CURRENTLY
  COMPARED. L WILL POINT TO THE ROOT OF C.
  THE NODE STRUCTURE OF FIG. 12a  IS ASSUMED.
  MEM(X) IMPLIES, THE CONTENTS AT ABSOLUTE
  ADDRESS X. }

 BEGIN
   {INITIALIZE}
   C:=LLINK(H);L:=H;
   USF:=FALSE;

   {SEARCH LOOP}
   WHILE C<>0 DO
    BEGIN
      CASE
        :IDENT(E)=IDENT(C):[USF:=TRUE;EXIT];
        :IDENT(E)>IDENT(C):L:=C+1;
        :IDENT(E)<IDENT(C):L:=C;
      END;
      C:=MEM(L);
    END;

   IF NOT USF THEN
    BEGIN
      MEM(L):=E;
      LLINK(E):=0;
      RLINK(E):=0;
    END;
 END;
```




(a)




(b)

Fig. 13: (a) Target machine architecture,
(b) Pipeline register bit assignment

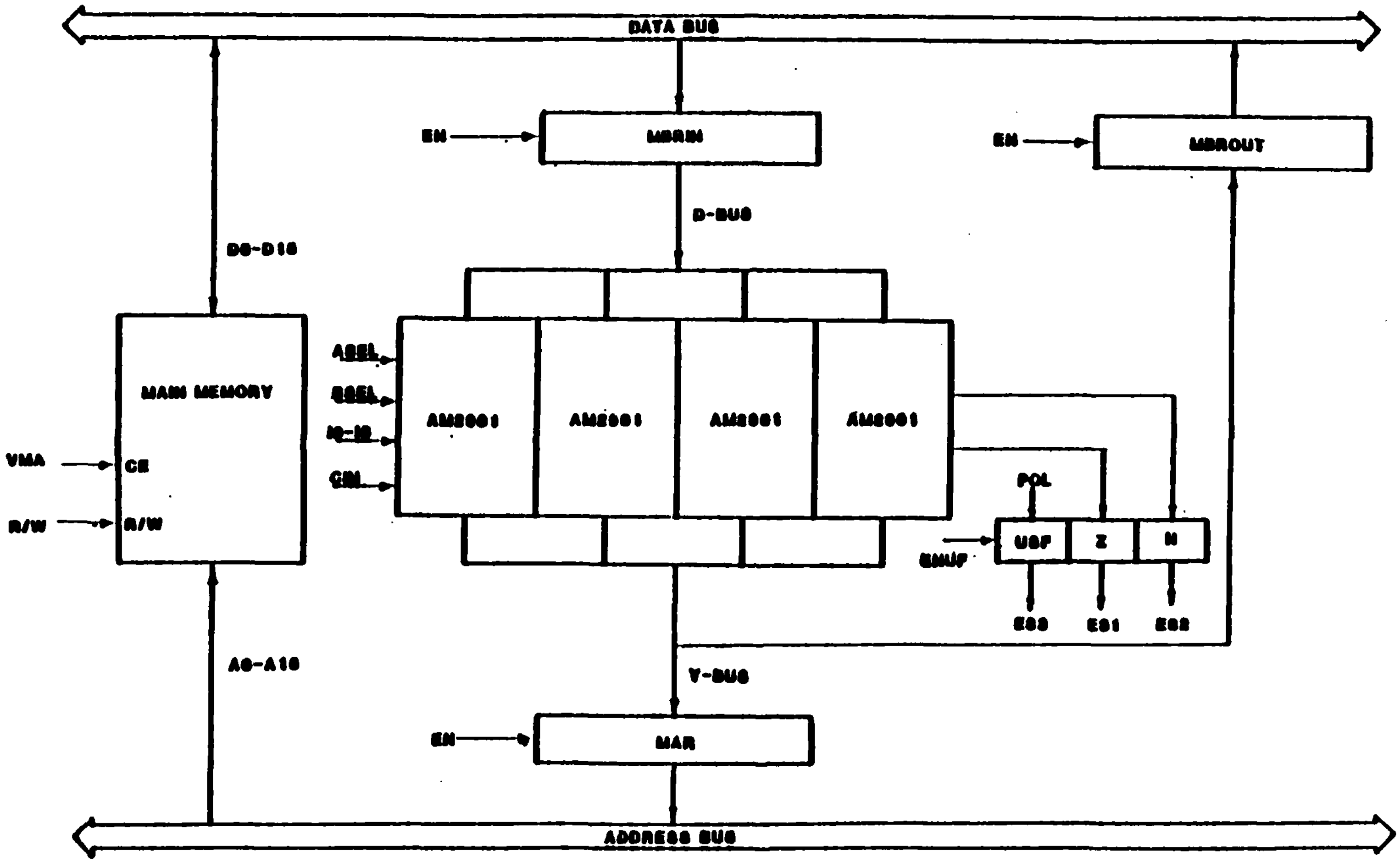



Fig. 14 : Processor Structure

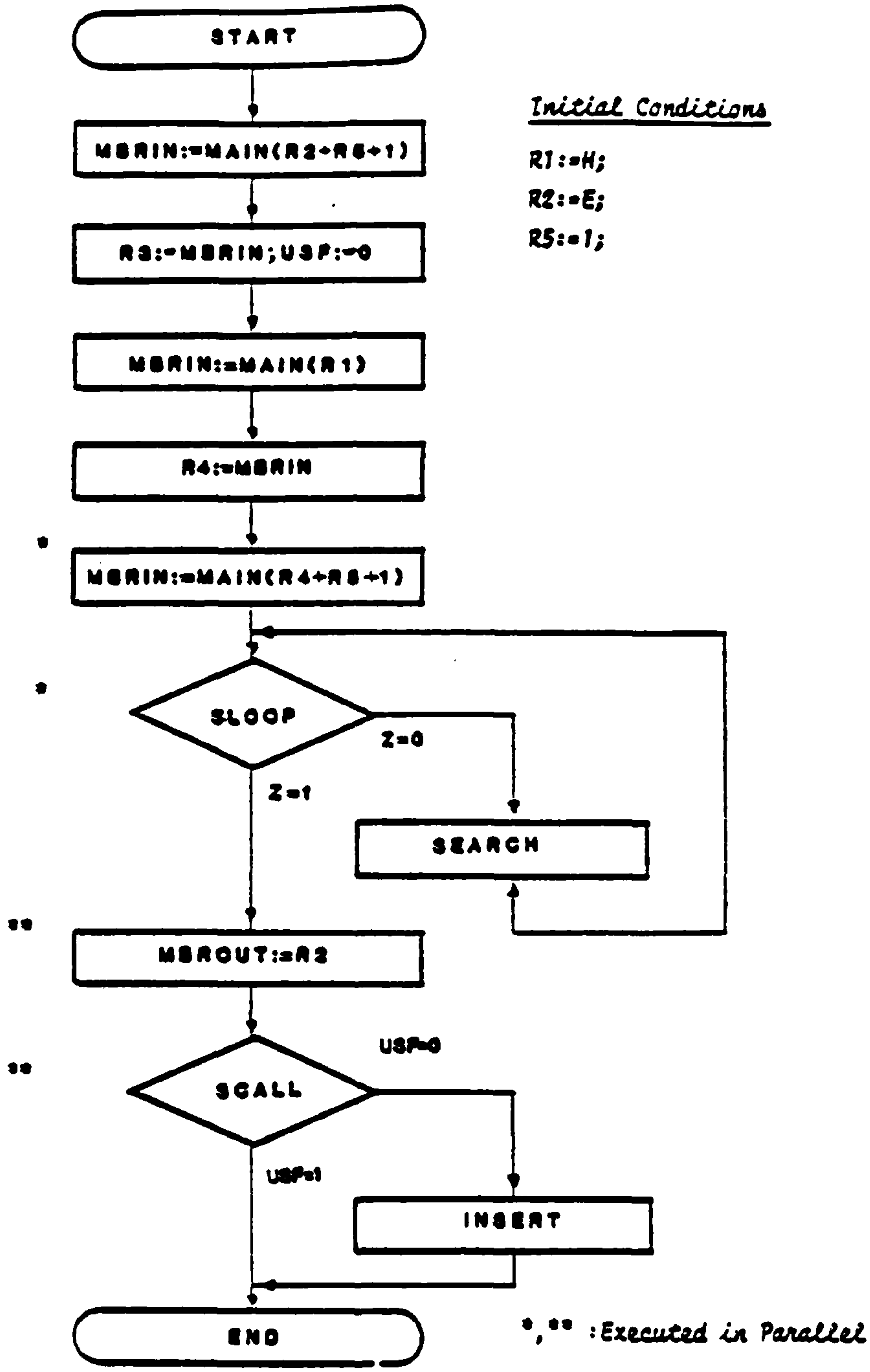
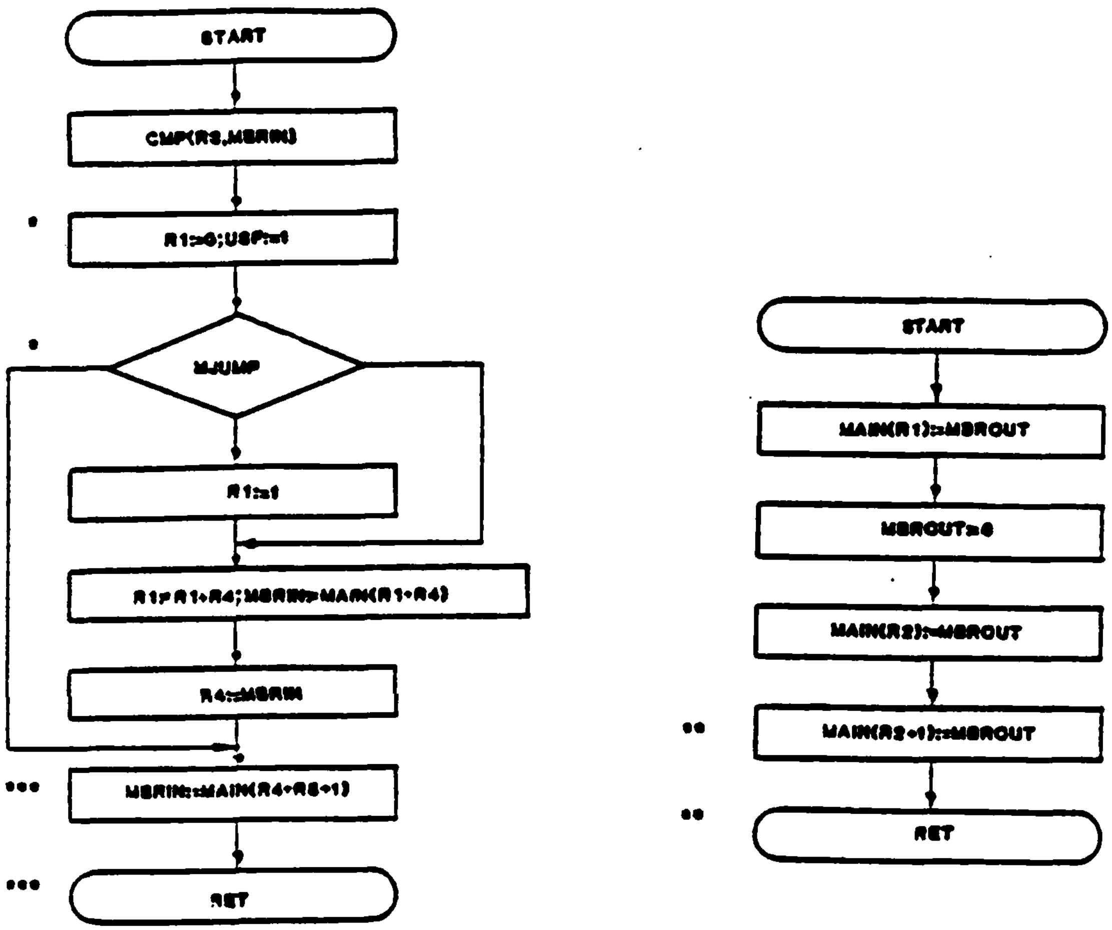



Fig. 15: Binary Tree Search and Insertion




(a) (b)

*Fig. 16: (a) Search Module, (b) Insert Module*

**,**,*** : Executed in Parallel*

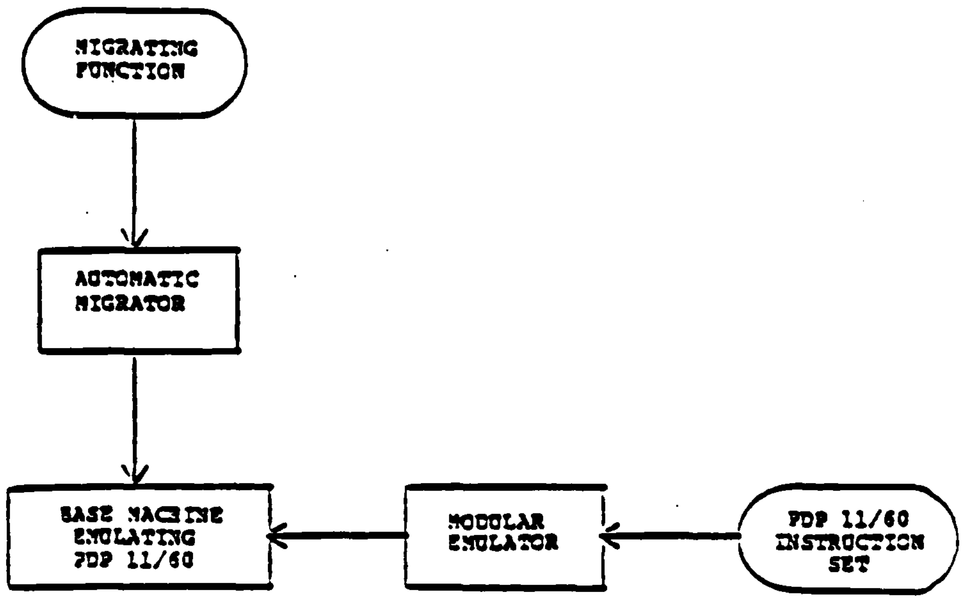



*Figure 17a . Basic Approach of the Scheme*

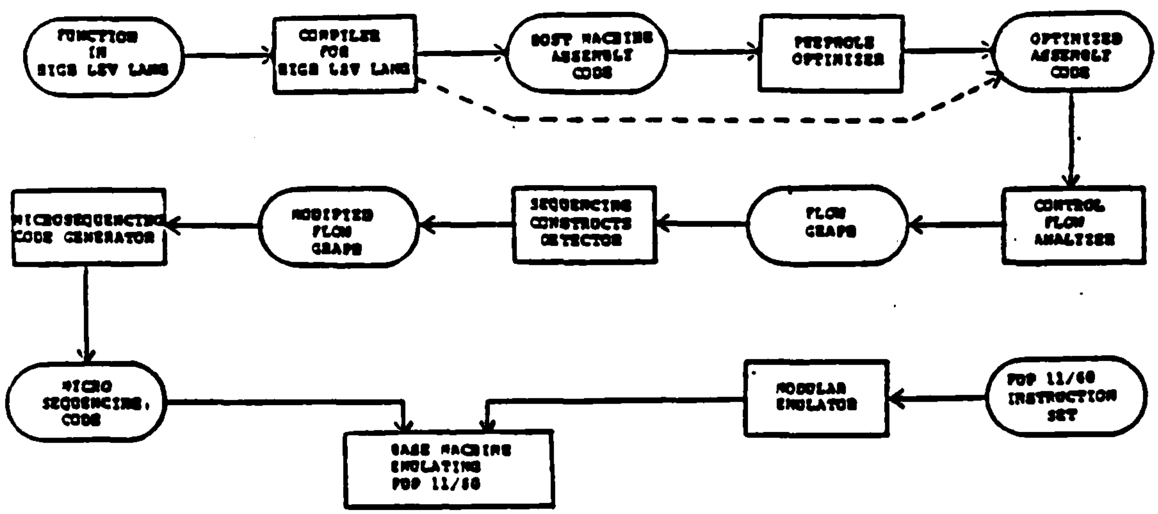



*Figure 17b . Various Phases in the Scheme for Automatic Migration*

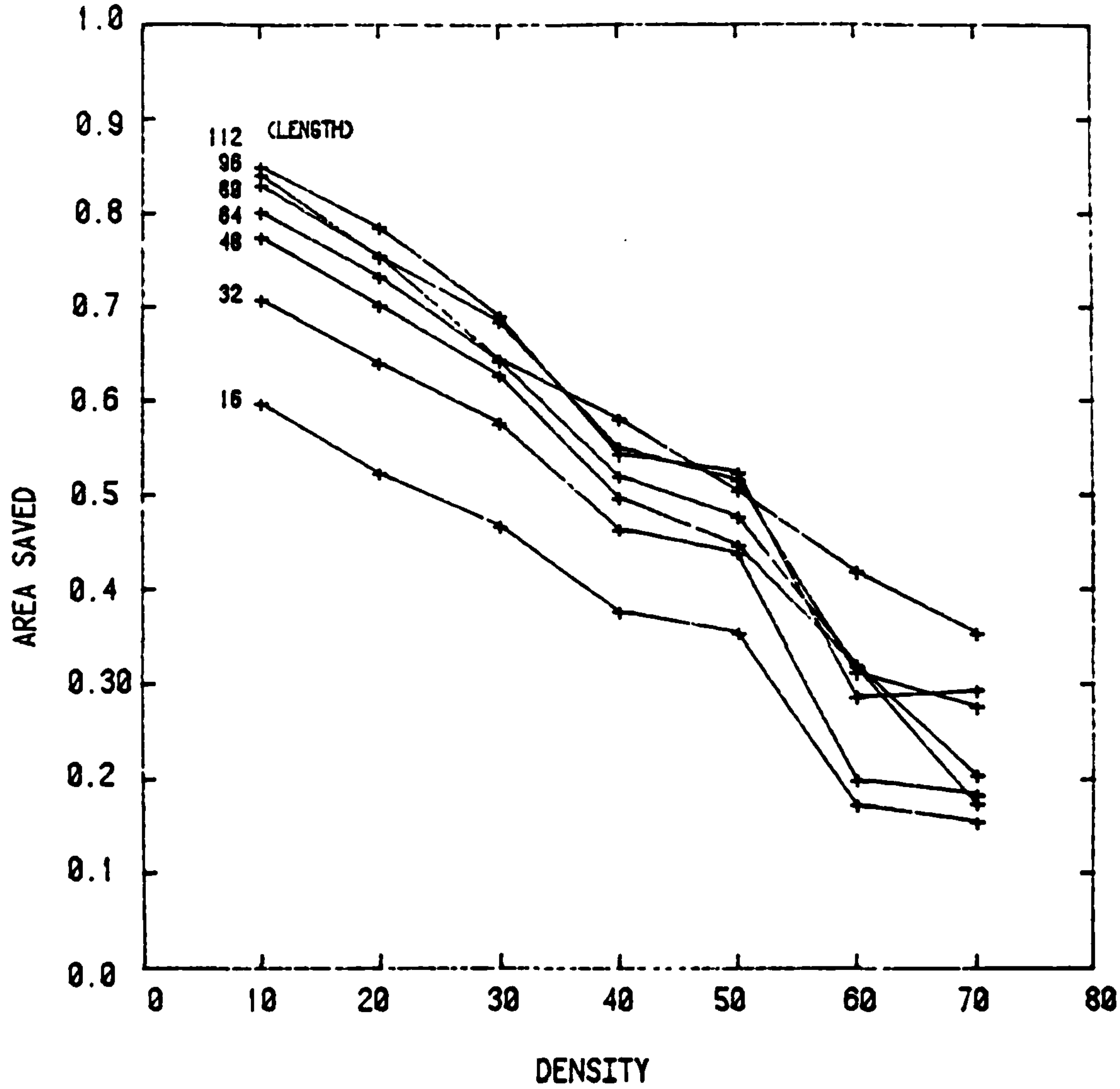



Fig. 18 **AREA SAVED vs. DENSITY**

| TRANSACTIONS | COMMENTS |
| --- | --- |
| JUMP | Unconditional Jump to a address |
| CALL | Unconditional call for a module |
| RTN | Conditional/Unconditional return |
| SJUMP | Binary conditional jump |
| SCALL | Binary conditional call |
| SLOOP | Binary conditional looping of a module |
| MJUMP | Multiway intra-module jump |
| MCALL | Multiway modular call |
| MLOOP | Multiway modular loop |
| DLOOP | Loop specified number of times |
| SBC | Push branch code into BC-stack |
| MAP | Branch to an externally mapped address |

Table 1: List of Firmware Transactions

# APPENDIX I

LINE BINARY TREE SEARCH AND INSERTION

```
            TITLE 'BINARY TREE SEARCH AND INSERTION'
;
;The following are the microsequencer transaction
;format definitions for a one-slice configuration
;of the proposed microcontrol scheme
;This control scheme is being used for our test
;example.
;
;Unconditional Jump
;
JUMP:  DEF     10R,4H01,10X
;
;Unconditional call
;
CALL:  DEF     10R,4H02,8V,8VX000
;
;Conditional or unconditional return
;
RET:   DEF     10R,4H03
;
;Binary conditional intra-module jump
;
BJUMP: DEF     10R,4H04,8VX,8VX
;
;Binary conditional call
;
BCALL: DEF     10R,4H05,8VX,8VX
;
;Binary Conditional Looping
;
BLOOP: DEF     10R,4H06,8VX,8VX
;
;Multiway intra-module jump
;
MJUMP: DEF     10R,4H07,8VX,8VX
;
;Multiway modular call
;
MCALL: DEF     10R,4H08,8VX,8VX
;
;Multiway modular loop
;
MLOOP:  DEF    10R,4H09,8VX,8VX
;
;Loop specified number of times
;
ILOOP: DEF     10R,4H0A
;
;Push branch code into BC-STACK
;
SBC:   DEF     10R,4H0B,16X
;
;Map an external address
```

# APPENDIX II

PLA CODE FORMATTER OUTPUT

| | | | |
|---|---|---|---|
| 0000000000000000 | 0000 | 0000000000010011 | |
| 0000000000000101 | 0001 | 0000000000000001 | |
| 0000000000000110 | 0011 | 0000000100000110 | |
| 0000000000001001 | 0101 | 0000000100000010 | 0000001100000100 |
| 0000000000001011 | 0100 | | 0000000100000011 |
| 0000000000001100 | 1101 | 1000010000000000 | 0000000110000000 |
| 0000000000001110 | 0111 | 0000000100000010 | 0000001100000100 |
| 0000000000010000 | 1101 | 0010000100000000 | |
| 0000000000010001 | 1000 | 0000010100000110 | |
| 0000000000010011 | 1010 | | |
| 0000000000010100 | 1101 | | 0000001010000000 |
| 0000000000010110 | 1001 | | 0000010000000011 |
| 0000000000011010 | 1011 | | |

END

FILMED

4-86

DTIC